(ƒ**asinh** *a*)
(ƒ**acosh** *a*) ▷ asinh *a*, acosh *a*, or atanh *a*, respectively.
(ƒ**atanh** *a*)

(ƒ**cis** *a*) ▷ Return $e^{i\,a} = \cos a + i \sin a$.

(ƒ**conjugate** *a*) ▷ Return complex conjugate of *a*.

(ƒ**max** *num*$^+$)
(ƒ**min** *num*$^+$) ▷ Greatest or least, respectively, of *num*s.

(ƒ{**round**|ƒ**fround**}
ƒ{**floor**|ƒ**ffloor**}
ƒ{**ceiling**|ƒ**fceiling**}
ƒ{**truncate**|ƒ**ftruncate**} *n* [*d*1])
▷ Return as **integer** or **float**, respectively, *n*/*d* rounded, or rounded towards $-\infty$, $+\infty$, or 0, respectively; and remainder.

(ƒ{**mod**|**rem**} *n d*)
▷ Same as ƒ**floor** or ƒ**truncate**, respectively, but return remainder only.

(ƒ**random** *limit* [*state*v**random-state**])
▷ Return non-negative random number less than *limit*, and of the same type.

(ƒ**make-random-state** [{*state*|NIL|T}NIL])
▷ Copy of **random-state** object *state* or of the current random state; or a randomly initialized fresh random state.

v***random-state*** ▷ Current random state.

(ƒ**float-sign** *num-a* [*num-b*1]) ▷ *num-b* with *num-a*'s sign.

(ƒ**signum** *n*)
▷ Number of magnitude 1 representing sign or phase of *n*.

(ƒ**numerator** *rational*)
(ƒ**denominator** *rational*)
▷ Numerator or denominator, respectively, of *rational*'s canonical form.

(ƒ**realpart** *number*)
(ƒ**imagpart** *number*)
▷ Real part or imaginary part, respectively, of *number*.

(ƒ**complex** *real* [*imag*0]) ▷ Make a complex number.

(ƒ**phase** *num*) ▷ Angle of *num*'s polar representation.

(ƒ**abs** *n*) ▷ Return $|n|$.

(ƒ**rational** *real*)
(ƒ**rationalize** *real*)
▷ Convert *real* to rational. Assume complete/limited accuracy for *real*.

(ƒ**float** *real* [*prototype*0.0F0])
▷ Convert *real* into float with type of *prototype*.

1.3 Logic Functions

Negative integers are used in two's complement representation.

(ƒ**boole** *operation int-a int-b*)
▷ Return value of bitwise logical *operation*. *operation*s are

c**boole-1** ▷ *int-a*.
c**boole-2** ▷ *int-b*.
c**boole-c1** ▷ ¬*int-a*.
c**boole-c2** ▷ ¬*int-b*.
c**boole-set** ▷ All bits set.
c**boole-clr** ▷ All bits zero.

Quick Reference

cl

Common
lisp

Bert Burgemeister

Contents

Typographic Conventions

name; $_f$**name**; $_g$**name**; $_m$**name**; $_s$**name**; $_v$***name****; $_c$**name**
▷ Symbol defined in Common Lisp; esp. function, generic function, macro, special operator, variable, constant.

them ▷ Placeholder for actual code.

`me` ▷ Literal text.

$[foo_{\boxed{\texttt{bar}}}]$ ▷ Either one *foo* or nothing; defaults to `bar`.

foo^*; $\{foo\}^*$ ▷ Zero or more *foo*s.

foo^+; $\{foo\}^+$ ▷ One or more *foo*s.

foos ▷ English plural denotes a list argument.

$\{foo \mid bar \mid baz\}$; $\begin{cases} foo \\ bar \\ baz \end{cases}$ ▷ Either *foo*, or *bar*, or *baz*.

$\left\{\begin{array}{|l} foo \\ bar \\ baz \end{array}\right.$ ▷ Anything from none to each of *foo*, *bar*, and *baz*.

$\widehat{foo}$ ▷ Argument *foo* is not evaluated.

$\widetilde{bar}$ ▷ Argument *bar* is possibly modified.

$foo^{\mathbb{P}_*}$ ▷ foo^* is evaluated as in $_s$**progn**; see page 20.

$\underline{foo}$; $\underline{bar}_2$; $\underline{baz}_n$ ▷ Primary, secondary, and nth return value.

`T`; `NIL` ▷ **t**, or truth in general; and **nil** or **()**.

1 Numbers

1.1 Predicates

($_f$= *number*$^+$)
($_f$/= *number*$^+$)
▷ T if all *number*s, or none, respectively, are equal in value.

($_f$> *number*$^+$)
($_f$>= *number*$^+$)
($_f$< *number*$^+$)
($_f$<= *number*$^+$)
▷ Return T if *number*s are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

($_f$**minusp** *a*)
($_f$**zerop** *a*)
($_f$**plusp** *a*)
▷ T if $a < 0$, $a = 0$, or $a > 0$, respectively.

($_f$**evenp** *int*)
($_f$**oddp** *int*)
▷ T if *int* is even or odd, respectively.

($_f$**numberp** *foo*)
($_f$**realp** *foo*)
($_f$**rationalp** *foo*)
($_f$**floatp** *foo*)
($_f$**integerp** *foo*)
($_f$**complexp** *foo*)
($_f$**random-state-p** *foo*)
▷ T if *foo* is of indicated type.

1.2 Numeric Functions

($_f$+ $a_{\boxed{0}}^*$)
($_f$* $a_{\boxed{1}}^*$)
▷ Return $\sum a$ or $\prod a$, respectively.

($_f$− *a* *b**)
($_f$/ *a* *b**)
▷ Return $a - \sum b$ or $a / \prod b$, respectively. Without any *b*s, return $-a$ or $1/a$, respectively.

($_f$**1+** *a*)
($_f$**1−** *a*)
▷ Return $a + 1$ or $a - 1$, respectively.

($\left\{\begin{array}{l} {}_m\textbf{incf} \\ {}_m\textbf{decf} \end{array}\right\}$ $\widetilde{place}$ [$delta_{\boxed{1}}$])
▷ Increment or decrement the value of *place* by *delta*. Return new value.

($_f$**exp** *p*)
($_f$**expt** *b* *p*)
▷ Return e^p or b^p, respectively.

($_f$**log** *a* [$b_{\boxed{e}}$])
▷ Return $\log_b a$ or, without *b*, $\ln a$.

($_f$**sqrt** *n*)
($_f$**isqrt** *n*)
▷ $\sqrt{n}$ in complex numbers/natural numbers.

($_f$**lcm** *integer*$^*_{\boxed{1}}$)
($_f$**gcd** *integer**)
▷ Least common multiple or greatest common denominator, respectively, of *integer*s. (**gcd**) returns 0.

$_c$**pi** ▷ **long-float** approximation of π, Ludolph's number.

($_f$**sin** *a*)
($_f$**cos** *a*)
($_f$**tan** *a*)
▷ $\sin a$, $\cos a$, or $\tan a$, respectively. (*a* in radians.)

($_f$**asin** *a*)
($_f$**acos** *a*)
▷ $\arcsin a$ or $\arccos a$, respectively, in radians.

($_f$**atan** *a* [$b_{\boxed{1}}$])
▷ $\arctan \frac{a}{b}$ in radians.

($_f$**sinh** *a*)
($_f$**cosh** *a*)
($_f$**tanh** *a*)
▷ $\sinh a$, $\cosh a$, or $\tanh a$, respectively.

(*f*char *string i*)
(*f*schar *string i*)
▷ Return zero-indexed *i*th character of string ignoring/obeying, respectively, fill pointer. setfable.

(*f*parse-integer *string* {:start *start*[0] | :end *end*[NIL] | :radix *int*[10] | :junk-allowed *bool*[NIL]})
▷ Return integer parsed from *string* and index[2] of parse end.

4 Conses

4.1 Predicates

(*f*consp *foo*)
(*f*listp *foo*)
▷ Return T if *foo* is of indicated type.

(*f*endp *list*)
(*f*null *foo*)
▷ Return T if *list*/*foo* is NIL.

(*f*atom *foo*)
▷ Return T if *foo* is not a cons.

(*f*tailp *foo list*)
▷ Return T if *foo* is a tail of *list*.

(*f*member *foo list* {:test *function*[#'eql] | :test-not *function* | :key *function*})
▷ Return tail of *list* starting with its first element matching *foo*. Return NIL if there is no such element.

({*f*member-if | *f*member-if-not} *test list* [:key *function*])
▷ Return tail of *list* starting with its first element satisfying *test*. Return NIL if there is no such element.

(*f*subsetp *list-a list-b* {:test *function*[#'eql] | :test-not *function* | :key *function*})
▷ Return T if *list-a* is a subset of *list-b*.

4.2 Lists

(*f*cons *foo bar*)
▷ Return new cons (*foo* . *bar*).

(*f*list *foo**)
▷ Return list of *foo*s.

(*f*list* *foo*⁺)
▷ Return list of *foo*s with last *foo* becoming cdr of last cons. Return *foo* if only one *foo* given.

(*f*make-list *num* [:initial-element *foo*[NIL]])
▷ New list with *num* elements set to *foo*.

(*f*list-length *list*)
▷ Length of *list*; NIL for circular *list*.

(*f*car *list*)
▷ Car of *list* or NIL if *list* is NIL. setfable.

(*f*cdr *list*)
(*f*rest *list*)
▷ Cdr of *list* or NIL if *list* is NIL. setfable.

(*f*nthcdr *n list*)
▷ Return tail of *list* after calling *f*cdr *n* times.

({*f*first | *f*second | *f*third | *f*fourth | *f*fifth | *f*sixth | ... | *f*ninth | *f*tenth} *list*)
▷ Return nth element of *list* if any, or NIL otherwise. setfable.

(*f*nth *n list*)
▷ Zero-indexed *n*th element of *list*. setfable.

(*f*c*X*r *list*)
▷ With *X* being one to four as and ds representing *f*cars and *f*cdrs, e.g. (*f*cadr *bar*) is equivalent to (*f*car (*f*cdr *bar*)). setfable.

(*f*last *list* [*num*[1]])
▷ Return list of last *num* conses of *list*.

*c*boole-eqv ▷ *int-a* ≡ *int-b*.
*c*boole-and ▷ *int-a* ∧ *int-b*.
*c*boole-andc1 ▷ ¬*int-a* ∧ *int-b*.
*c*boole-andc2 ▷ *int-a* ∧ ¬*int-b*.
*c*boole-nand ▷ ¬(*int-a* ∧ *int-b*).
*c*boole-ior ▷ *int-a* ∨ *int-b*.
*c*boole-orc1 ▷ ¬*int-a* ∨ *int-b*.
*c*boole-orc2 ▷ *int-a* ∨ ¬*int-b*.
*c*boole-xor ▷ ¬(*int-a* ≡ *int-b*).
*c*boole-nor ▷ ¬(*int-a* ∨ *int-b*).

(*f*lognot *integer*) ▷ ¬*integer*.

(*f*logeqv *integer**)
(*f*logand *integer**)
▷ Return value of exclusive-nored or anded *integer*s, respectively. Without any *integer*, return −1.

(*f*logandc1 *int-a int-b*) ▷ ¬*int-a* ∧ *int-b*.

(*f*logandc2 *int-a int-b*) ▷ *int-a* ∧ ¬*int-b*.

(*f*lognand *int-a int-b*) ▷ ¬(*int-a* ∧ *int-b*).

(*f*logxor *integer**)
(*f*logior *integer**)
▷ Return value of exclusive-ored or ored *integer*s, respectively. Without any *integer*, return 0.

(*f*logorc1 *int-a int-b*) ▷ ¬*int-a* ∨ *int-b*.

(*f*logorc2 *int-a int-b*) ▷ *int-a* ∨ ¬*int-b*.

(*f*lognor *int-a int-b*) ▷ ¬(*int-a* ∨ *int-b*).

(*f*logbitp *i int*) ▷ T if zero-indexed *i*th bit of *int* is set.

(*f*logtest *int-a int-b*)
▷ Return T if there is any bit set in *int-a* which is set in *int-b* as well.

(*f*logcount *int*)
▷ Number of 1 bits in *int* ≥ 0, number of 0 bits in *int* < 0.

1.4 Integer Functions

(*f*integer-length *integer*)
▷ Number of bits necessary to represent *integer*.

(*f*ldb-test *byte-spec integer*)
▷ Return T if any bit specified by *byte-spec* in *integer* is set.

(*f*ash *integer count*)
▷ Return copy of *integer* arithmetically shifted left by *count* adding zeros at the right, or, for *count* < 0, shifted right discarding bits.

(*f*ldb *byte-spec integer*)
▷ Extract byte denoted by *byte-spec* from *integer*. setfable.

({*f*deposit-field | *f*dpb} *int-a byte-spec int-b*)
▷ Return *int-b* with bits denoted by *byte-spec* replaced by corresponding bits of *int-a*, or by the low (*f*byte-size *byte-spec*) bits of *int-a*, respectively.

(*f*mask-field *byte-spec integer*)
▷ Return copy of *integer* with all bits unset but those denoted by *byte-spec*. setfable.

(*f*byte *size position*)
▷ Byte specifier for a byte of *size* bits starting at a weight of $2^{position}$.

(*f*byte-size *byte-spec*)
(*f*byte-position *byte-spec*)
▷ Size or position, respectively, of *byte-spec*.

1.5 Implementation-Dependent

$_c$**short-float**, $_c$**single-float**, $_c$**double-float**, $_c$**long-float** – {**epsilon**, **negative-epsilon**}
▷ Smallest possible number making a difference when added or subtracted, respectively.

$_c$**least-negative**, $_c$**least-negative-normalized**, $_c$**least-positive**, $_c$**least-positive-normalized** – {**short-float**, **single-float**, **double-float**, **long-float**}
▷ Available numbers closest to -0 or $+0$, respectively.

$_c$**most-negative**, $_c$**most-positive** – {**short-float**, **single-float**, **double-float**, **long-float**, **fixnum**}
▷ Available numbers closest to $-\infty$ or $+\infty$, respectively.

($_f$**decode-float** *n*)
($_f$**integer-decode-float** *n*)
▷ Return significand, exponent, and sign of **float** *n*.

($_f$**scale-float** *n* [*i*]) ▷ With *n*'s radix *b*, return nb^i.

($_f$**float-radix** *n*)
($_f$**float-digits** *n*)
($_f$**float-precision** *n*)
▷ Radix, number of digits in that radix, or precision in that radix, respectively, of float *n*.

($_f$**upgraded-complex-part-type** *foo* [*environment*$_{\texttt{NIL}}$])
▷ Type of most specialized **complex** number able to hold parts of type *foo*.

2 Characters

The **standard-char** type comprises `a-z`, `A-Z`, `0-9`, `Newline`, `Space`, and `!?$"'‘.:,;*+-/|\~_^<=>#%@&()[]{}`.

($_f$**characterp** *foo*)
($_f$**standard-char-p** *char*) ▷ T if argument is of indicated type.

($_f$**graphic-char-p** *character*)
($_f$**alpha-char-p** *character*)
($_f$**alphanumericp** *character*)
▷ T if *character* is visible, alphabetic, or alphanumeric, respectively.

($_f$**upper-case-p** *character*)
($_f$**lower-case-p** *character*)
($_f$**both-case-p** *character*)
▷ Return T if *character* is uppercase, lowercase, or able to be in another case, respectively.

($_f$**digit-char-p** *character* [*radix*$_{10}$])
▷ Return its weight if *character* is a digit, or NIL otherwise.

($_f$**char=** *character*$^+$)
($_f$**char/=** *character*$^+$)
▷ Return T if all *character*s, or none, respectively, are equal.

($_f$**char-equal** *character*$^+$)
($_f$**char-not-equal** *character*$^+$)
▷ Return T if all *character*s, or none, respectively, are equal ignoring case.

($_f$**char>** *character*$^+$)
($_f$**char>=** *character*$^+$)
($_f$**char<** *character*$^+$)
($_f$**char<=** *character*$^+$)
▷ Return T if *character*s are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

($_f$**char-greaterp** *character*$^+$)
($_f$**char-not-lessp** *character*$^+$)
($_f$**char-lessp** *character*$^+$)
($_f$**char-not-greaterp** *character*$^+$)
▷ Return T if *character*s are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.

($_f$**char-upcase** *character*)
($_f$**char-downcase** *character*)
▷ Return corresponding uppercase/lowercase character, respectively.

($_f$**digit-char** *i* [*radix*$_{10}$]) ▷ Character representing digit *i*.

($_f$**char-name** *char*) ▷ *char*'s name if any, or NIL.

($_f$**name-char** *foo*) ▷ Character named *foo* if any, or NIL.

($_f$**char-int** *character*)
($_f$**char-code** *character*) ▷ Code of *character*.

($_f$**code-char** *code*) ▷ Character with *code*.

$_c$**char-code-limit** ▷ Upper bound of ($_f$**char-code** *char*); ≥ 96.

($_f$**character** *c*) ▷ Return #*c*.

3 Strings

Strings can as well be manipulated by array and sequence functions; see pages 10 and 12.

($_f$**stringp** *foo*)
($_f$**simple-string-p** *foo*) ▷ T if *foo* is of indicated type.

({$_f$**string=**, $_f$**string-equal**} *foo bar* {**:start1** *start-foo*$_0$, **:start2** *start-bar*$_0$, **:end1** *end-foo*$_{\texttt{NIL}}$, **:end2** *end-bar*$_{\texttt{NIL}}$})
▷ Return T if subsequences of *foo* and *bar* are equal. Obey/ignore, respectively, case.

({$_f$**string{/=**|**-not-equal}**, $_f$**string{>**|**-greaterp}**, $_f$**string{>=**|**-not-lessp}**, $_f$**string{<**|**-lessp}**, $_f$**string{<=**|**-not-greaterp}**} *foo bar* {**:start1** *start-foo*$_0$, **:start2** *start-bar*$_0$, **:end1** *end-foo*$_{\texttt{NIL}}$, **:end2** *end-bar*$_{\texttt{NIL}}$})
▷ If *foo* is lexicographically not equal, greater, not less, less, or not greater, respectively, then return position of first mismatching character in *foo*. Otherwise return NIL. Obey/ignore, respectively, case.

($_f$**make-string** *size* {**:initial-element** *char*, **:element-type** *type*$_{\texttt{character}}$})
▷ Return string of length *size*.

($_f$**string** *x*)
({$_f$**string-capitalize**, $_f$**string-upcase**, $_f$**string-downcase**} *x* {**:start** *start*$_0$, **:end** *end*$_{\texttt{NIL}}$})
▷ Convert *x* (**symbol**, **string**, or **character**) into a string, a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

({$_f$**nstring-capitalize**, $_f$**nstring-upcase**, $_f$**nstring-downcase**} $\widetilde{\textit{string}}$ {**:start** *start*$_0$, **:end** *end*$_{\texttt{NIL}}$})
▷ Convert *string* into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

({$_f$**string-trim**, $_f$**string-left-trim**, $_f$**string-right-trim**} *char-bag string*)
▷ Return *string* with all characters in sequence *char-bag* removed from both ends, from the beginning, or from the end, respectively.

6 Sequences

6.1 Sequence Predicates

(f{**every** **notevery**} *test sequence*+)
▷ Return NIL or T, respectively, as soon as *test* on any set of corresponding elements of *sequence*s returns NIL.

(f{**some** **notany**} *test sequence*+)
▷ Return value of *test* or NIL, respectively, as soon as *test* on any set of corresponding elements of *sequence*s returns non-NIL.

(f**mismatch** *sequence-a sequence-b* {**:from-end** *bool*NIL | {**:test** *function*#'eql | **:test-not** *function*} | **:start1** *start-a*0 | **:start2** *start-b*0 | **:end1** *end-a*NIL | **:end2** *end-b*NIL | **:key** *function*})
▷ Return position in *sequence-a* where *sequence-a* and *sequence-b* begin to mismatch. Return NIL if they match entirely.

6.2 Sequence Functions

(f**make-sequence** *sequence-type size* [**:initial-element** *foo*])
▷ Make sequence of *sequence-type* with *size* elements.

(f**concatenate** *type sequence**)
▷ Return concatenated sequence of *type*.

(f**merge** *type* ~*sequence*-*a* ~*sequence*-*b test* [**:key** *function*NIL])
▷ Return interleaved sequence of *type*. Merged sequence will be sorted if both *sequence-a* and *sequence-b* are sorted.

(f**fill** ~*sequence foo* {**:start** *start*0 | **:end** *end*NIL})
▷ Return *sequence* after setting elements between *start* and *end* to *foo*.

(f**length** *sequence*)
▷ Return length of *sequence* (being value of fill pointer if applicable).

(f**count** *foo sequence* {**:from-end** *bool*NIL | {**:test** *function*#'eql | **:test-not** *function*} | **:start** *start*0 | **:end** *end*NIL | **:key** *function*})
▷ Return number of elements in *sequence* which match *foo*.

(f{**count-if** **count-if-not**} *test sequence* {**:from-end** *bool*NIL | **:start** *start*0 | **:end** *end*NIL | **:key** *function*})
▷ Return number of elements in *sequence* which satisfy *test*.

(f**elt** *sequence index*)
▷ Return element of *sequence* pointed to by zero-indexed *index*. **setf**able.

(f**subseq** *sequence start* [*end*NIL])
▷ Return subsequence of *sequence* between *start* and *end*. **setf**able.

(f{**sort** **stable-sort**} ~*sequence test* [**:key** *function*])
▷ Return *sequence* sorted. Order of elements considered equal is not guaranteed/retained, respectively.

(f**reverse** *sequence*)
(f**nreverse** ~*sequence*) ▷ Return *sequence* in reverse order.

(f{**butlast** *list* **nbutlast** ~*list*} [*num*1]) ▷ *list* excluding last *num* conses.

(f{**rplaca** **rplacd**} ~*cons object*)
▷ Replace car, or cdr, respectively, of *cons* with *object*.

(f**ldiff** *list foo*)
▷ If *foo* is a tail of *list*, return preceding part of *list*. Otherwise return *list*.

(f**adjoin** *foo list* {{**:test** *function*#'eql | **:test-not** *function*} | **:key** *function*})
▷ Return *list* if *foo* is already member of *list*. If not, return (f**cons** *foo list*).

(m**pop** ~*place*)
▷ Set *place* to (f**cdr** *place*), return (f**car** *place*).

(m**push** *foo* ~*place*) ▷ Set *place* to (f**cons** *foo place*).

(m**pushnew** *foo* ~*place* {{**:test** *function*#'eql | **:test-not** *function*} | **:key** *function*})
▷ Set *place* to (f**adjoin** *foo place*).

(f**append** [*proper-list** *foo*NIL])
(f**nconc** [*non-circular-list** *foo*NIL])
▷ Return concatenated list or, with only one argument, *foo*. *foo* can be of any type.

(f**revappend** *list foo*)
(f**nreconc** ~*list foo*)
▷ Return concatenated list after reversing order in *list*.

(f{**mapcar** **maplist**} *function list*+)
▷ Return list of return values of *function* successively invoked with corresponding arguments, either cars or cdrs, respectively, from each *list*.

(f{**mapcan** **mapcon**} *function* ~*list*+)
▷ Return list of concatenated return values of *function* successively invoked with corresponding arguments, either cars or cdrs, respectively, from each *list*. *function* should return a list.

(f{**mapc** **mapl**} *function list*+)
▷ Return first *list* after successively applying *function* to corresponding arguments, either cars or cdrs, respectively, from each *list*. *function* should have some side effects.

(f**copy-list** *list*) ▷ Return copy of *list* with shared elements.

4.3 Association Lists

(f**pairlis** *keys values* [*alist*NIL])
▷ Prepend to *alist* an association list made from lists *keys* and *values*.

(f**acons** *key value alist*)
▷ Return *alist* with a (*key* . *value*) pair added.

(f{**assoc** **rassoc**} *foo alist* {{**:test** *test*#'eql | **:test-not** *test*} | **:key** *function*})
(f{**assoc-if**[**-not**] **rassoc-if**[**-not**]} *test alist* [**:key** *function*])
▷ First cons whose car, or cdr, respectively, satisfies *test*.

(f**copy-alist** *alist*) ▷ Return copy of *alist*.

4.4 Trees

(ƒ**tree-equal** *foo bar* {**:test** *test*[#'eql] / **:test-not** *test*})
▷ Return T if trees *foo* and *bar* have same shape and leaves satisfying *test*.

({ƒ**subst** *new old tree* / ƒ**nsubst** *new old tree*} {| **:test** *function*[#'eql] / **:test-not** *function* / **:key** *function* |})
▷ Make copy of *tree* with each subtree or leaf matching *old* replaced by *new*.

({ƒ**subst-if**[**-not**] *new test tree* / ƒ**nsubst-if**[**-not**] *new test tree*} [**:key** *function*])
▷ Make copy of *tree* with each subtree or leaf satisfying *test* replaced by *new*.

({ƒ**sublis** *association-list tree* / ƒ**nsublis** *association-list tree*} {| **:test** *function*[#'eql] / **:test-not** *function* / **:key** *function* |})
▷ Make copy of *tree* with each subtree or leaf matching a key in *association-list* replaced by that key's value.

(ƒ**copy-tree** *tree*) ▷ Copy of *tree* with same shape and leaves.

4.5 Sets

({ƒ**intersection** / ƒ**set-difference** / ƒ**union** / ƒ**set-exclusive-or**} *a b*; {ƒ**nintersection** / ƒ**nset-difference**} *a b*; {ƒ**nunion** / ƒ**nset-exclusive-or**} *a b* {| **:test** *function*[#'eql] / **:test-not** *function* / **:key** *function* |})
▷ Return $a \cap b$, $a \setminus b$, $a \cup b$, or $a \triangle b$, respectively, of lists *a* and *b*.

5 Arrays

5.1 Predicates

(ƒ**arrayp** *foo*)
(ƒ**vectorp** *foo*)
(ƒ**simple-vector-p** *foo*) ▷ T if *foo* is of indicated type.
(ƒ**bit-vector-p** *foo*)
(ƒ**simple-bit-vector-p** *foo*)

(ƒ**adjustable-array-p** *array*)
(ƒ**array-has-fill-pointer-p** *array*)
▷ T if *array* is adjustable/has a fill pointer, respectively.

(ƒ**array-in-bounds-p** *array* [*subscripts*])
▷ Return T if *subscripts* are in *array*'s bounds.

5.2 Array Functions

({ƒ**make-array** *dimension-sizes* [**:adjustable** *bool*[NIL]] / ƒ**adjust-array** *array dimension-sizes*} {| **:element-type** *type*[T] / **:fill-pointer** {*num* | *bool*}[NIL] / {**:initial-element** *obj* / **:initial-contents** *tree-or-array* / **:displaced-to** *array*[NIL] [**:displaced-index-offset** *i*[0]]} |})
▷ Return fresh, or readjust, respectively, vector or array.

(ƒ**aref** *array* [*subscripts*])
▷ Return array element pointed to by *subscripts*. **setf**able.

(ƒ**row-major-aref** *array i*)
▷ Return *i*th element of *array* in row-major order. **setf**able.

(ƒ**array-row-major-index** *array* [*subscripts*])
▷ Index in row-major order of the element denoted by *subscripts*.

(ƒ**array-dimensions** *array*)
▷ List containing the lengths of *array*'s dimensions.

(ƒ**array-dimension** *array i*)
▷ Length of *i*th dimension of *array*.

(ƒ**array-total-size** *array*) ▷ Number of elements in *array*.

(ƒ**array-rank** *array*) ▷ Number of dimensions of *array*.

(ƒ**array-displacement** *array*) ▷ Target array and offset.

(ƒ**bit** *bit-array* [*subscripts*])
(ƒ**sbit** *simple-bit-array* [*subscripts*])
▷ Return element of *bit-array* or of *simple-bit-array*. **setf**able.

(ƒ**bit-not** *bit-array* [*result-bit-array*[NIL]])
▷ Return result of bitwise negation of *bit-array*. If *result-bit-array* is T, put result in *bit-array*; if it is NIL, make a new array for result.

({ƒ**bit-eqv** / ƒ**bit-and** / ƒ**bit-andc1** / ƒ**bit-andc2** / ƒ**bit-nand** / ƒ**bit-ior** / ƒ**bit-orc1** / ƒ**bit-orc2** / ƒ**bit-xor** / ƒ**bit-nor**} *bit-array-a bit-array-b* [*result-bit-array*[NIL]])
▷ Return result of bitwise logical operations (cf. operations of ƒ**boole**, page 4) on *bit-array-a* and *bit-array-b*. If *result-bit-array* is T, put result in *bit-array-a*; if it is NIL, make a new array for result.

c**array-rank-limit** ▷ Upper bound of array rank; ≥ 8.

c**array-dimension-limit**
▷ Upper bound of an array dimension; ≥ 1024.

c**array-total-size-limit** ▷ Upper bound of array size; ≥ 1024.

5.3 Vector Functions

Vectors can as well be manipulated by sequence functions; see section 6.

(ƒ**vector** *foo**) ▷ Return fresh simple vector of *foo*s.

(ƒ**svref** *vector i*) ▷ Element *i* of simple *vector*. **setf**able.

(ƒ**vector-push** *foo vector*)
▷ Return NIL if *vector*'s fill pointer equals size of *vector*. Otherwise replace element of *vector* pointed to by fill pointer with *foo*; then increment fill pointer.

(ƒ**vector-push-extend** *foo vector* [*num*])
▷ Replace element of *vector* pointed to by fill pointer with *foo*, then increment fill pointer. Extend *vector*'s size by $\geq$ *num* if necessary.

(ƒ**vector-pop** *vector*)
▷ Return element of *vector* its fillpointer points to after decrementation.

(ƒ**fill-pointer** *vector*) ▷ Fill pointer of *vector*. **setf**able.

(ₓ**fboundp** {*foo* | (**setf** *foo*)}) ▷ T if *foo* is a global function or macro.

9.2 Variables

({ₘ**defconstant** | ₘ**defparameter**} *foo form* [*doc*])
▷ Assign value of *form* to global constant/dynamic variable *foo*.

(ₘ**defvar** *foo* [*form* [*doc*]])
▷ Unless bound already, assign value of *form* to dynamic variable *foo*.

({ₘ**setf** | ₘ**psetf**} {*place form*}*)
▷ Set *place*s to primary values of *form*s. Return values of last *form*/NIL; work sequentially/in parallel, respectively.

({ₛ**setq** | ₘ**psetq**} {*symbol form*}*)
▷ Set *symbol*s to primary values of *form*s. Return value of last *form*/NIL; work sequentially/in parallel, respectively.

(ₓ**set** *symbol foo*) ▷ Set *symbol*'s value cell to *foo*. Deprecated.

(ₘ**multiple-value-setq** *vars form*)
▷ Set elements of *vars* to the values of *form*. Return *form*'s primary value.

(ₘ**shiftf** *place*⁺ *foo*)
▷ Store value of *foo* in rightmost *place* shifting values of *place*s left, returning first *place*.

(ₘ**rotatef** *place**)
▷ Rotate values of *place*s left, old first becoming new last *place*'s value. Return NIL.

(ₓ**makunbound** *foo*) ▷ Delete special variable *foo* if any.

(ₓ**get** *symbol key* [*default*NIL])
(ₓ**getf** *place key* [*default*NIL])
▷ First entry *key* from property list stored in *symbol*/in *place*, respectively, or *default* if there is no *key*. **setf**able.

(ₓ**get-properties** *property-list keys*)
▷ Return key and value of first entry from *property-list* matching a key from *keys*, and tail of *property-list* starting with that key. Return NIL, NIL, and NIL if there was no matching key in *property-list*.
(2) (3) (3)

(ₓ**remprop** *symbol key*)
(ₘ**remf** *place key*)
▷ Remove first entry *key* from property list stored in *symbol*/in *place*, respectively. Return T if *key* was there, or NIL otherwise.

(ₛ**progv** *symbols values form*ᴾ*)
▷ Evaluate *form*s with locally established dynamic bindings of *symbols* to *values* or NIL. Return values of *form*s.

({ₛ**let** | ₛ**let***} ({*name* | (*name* [*value*NIL])}*) (**declare** *decl**)* *form*ᴾ*)
▷ Evaluate *form*s with *name*s lexically bound (in parallel or sequentially, respectively) to *value*s. Return values of *form*s.

(ₘ**multiple-value-bind** (*var**) *values-form* (**declare** *decl**)*
*body-form*ᴾ*)
▷ Evaluate *body-form*s with *var*s lexically bound to the return values of *values-form*. Return values of *body-form*s.

(ₘ**destructuring-bind** *destruct-λ bar* (**declare** *decl**)* *form*ᴾ*)
▷ Evaluate *form*s with variables from tree *destruct-λ* bound to corresponding elements of tree *bar*, and return their values. *destruct-λ* resembles *macro-λ* (section 9.4), but without any **&environment** clause.

({ₓ**find** | ₓ**position**} *foo sequence* {**:from-end** *bool*NIL | {**:test** *function*#'eql | **:test-not** *test*} | **:start** *start*0 | **:end** *end*NIL | **:key** *function*})
▷ Return first element in *sequence* which matches *foo*, or its position relative to the begin of *sequence*, respectively.

({ₓ**find-if** | ₓ**find-if-not** | ₓ**position-if** | ₓ**position-if-not**} *test sequence* {**:from-end** *bool*NIL | **:start** *start*0 | **:end** *end*NIL | **:key** *function*})
▷ Return first element in *sequence* which satisfies *test*, or its position relative to the begin of *sequence*, respectively.

(ₓ**search** *sequence-a sequence-b* {**:from-end** *bool*NIL | {**:test** *function*#'eql | **:test-not** *function*} | **:start1** *start-a*0 | **:start2** *start-b*0 | **:end1** *end-a*NIL | **:end2** *end-b*NIL | **:key** *function*})
▷ Search *sequence-b* for a subsequence matching *sequence-a*. Return position in *sequence-b*, or NIL.

({ₓ**remove** *foo sequence* | ₓ**delete** *foo sequence*} {**:from-end** *bool*NIL | {**:test** *function*#'eql | **:test-not** *function*} | **:start** *start*0 | **:end** *end*NIL | **:key** *function* | **:count** *count*NIL})
▷ Make copy of *sequence* without elements matching *foo*.

({ₓ**remove-if** | ₓ**remove-if-not** | ₓ**delete-if** | ₓ**delete-if-not**} *test sequence*} {**:from-end** *bool*NIL | **:start** *start*0 | **:end** *end*NIL | **:key** *function* | **:count** *count*NIL})
▷ Make copy of *sequence* with all (or *count*) elements satisfying *test* removed.

({ₓ**remove-duplicates** *sequence* | ₓ**delete-duplicates** *sequence*} {**:from-end** *bool*NIL | {**:test** *function*#'eql | **:test-not** *function*} | **:start** *start*0 | **:end** *end*NIL | **:key** *function*})
▷ Make copy of *sequence* without duplicates.

({ₓ**substitute** *new old sequence* | ₓ**nsubstitute** *new old sequence*} {**:from-end** *bool*NIL | {**:test** *function*#'eql | **:test-not** *function*} | **:start** *start*0 | **:end** *end*NIL | **:key** *function* | **:count** *count*NIL})
▷ Make copy of *sequence* with all (or *count*) *old*s replaced by *new*.

({ₓ**substitute-if** | ₓ**substitute-if-not** | ₓ**nsubstitute-if** | ₓ**nsubstitute-if-not**} *new test sequence*} {**:from-end** *bool*NIL | **:start** *start*0 | **:end** *end*NIL | **:key** *function* | **:count** *count*NIL})
▷ Make copy of *sequence* with all (or *count*) elements satisfying *test* replaced by *new*.

(ₓ**replace** *sequence-a sequence-b* {**:start1** *start-a*0 | **:start2** *start-b*0 | **:end1** *end-a*NIL | **:end2** *end-b*NIL})
▷ Replace elements of *sequence-a* with elements of *sequence-b*.

(ƒ**map** *type function sequence*$^+$)
▷ Apply *function* successively to corresponding elements of the *sequences*. Return values as a sequence of *type*. If *type* is `NIL`, return `NIL`.

(ƒ**map-into** *result-sequence function sequence**)
▷ Store into *result-sequence* successively values of *function* applied to corresponding elements of the *sequences*.

(ƒ**reduce** *function sequence* { **:initial-value** *foo*[NIL] | **:from-end** *bool*[NIL] | **:start** *start*[0] | **:end** *end*[NIL] | **:key** *function* })
▷ Starting with the first two elements of *sequence*, apply *function* successively to its last return value together with the next element of *sequence*. Return last value of function.

(ƒ**copy-seq** *sequence*)
▷ Copy of *sequence* with shared elements.

7 Hash Tables

The Loop Facility provides additional hash table-related functionality; see **loop**, page 21.

Key-value storage similar to hash tables can as well be achieved using association lists and property lists; see pages 9 and 16.

(ƒ**hash-table-p** *foo*) ▷ Return `T` if *foo* is of type **hash-table**.

(ƒ**make-hash-table** { **:test** {ƒ**eq**|ƒ**eql**|ƒ**equal**|ƒ**equalp**}[#'eql] | **:size** *int* | **:rehash-size** *num* | **:rehash-threshold** *num* })
▷ Make a hash table.

(ƒ**gethash** *key hash-table* [*default*[NIL]])
▷ Return object with *key* if any or *default* otherwise; and `T` if found, `NIL` otherwise. **setf**able.

(ƒ**hash-table-count** *hash-table*)
▷ Number of entries in *hash-table*.

(ƒ**remhash** *key hash-table*)
▷ Remove from *hash-table* entry with *key* and return `T` if it existed. Return `NIL` otherwise.

(ƒ**clrhash** *hash-table*) ▷ Empty *hash-table*.

(ƒ**maphash** *function hash-table*)
▷ Iterate over *hash-table* calling *function* on key and value. Return `NIL`.

(*m*with-hash-table-iterator (*foo hash-table*) (**declare** *decl**)* *form*^{P*})
▷ Return values of *forms*. In *forms*, invocations of (*foo*) return: `T` if an entry is returned; its key; its value.

(ƒ**hash-table-test** *hash-table*)
▷ Test function used in *hash-table*.

(ƒ**hash-table-size** *hash-table*)
(ƒ**hash-table-rehash-size** *hash-table*)
(ƒ**hash-table-rehash-threshold** *hash-table*)
▷ Current size, rehash-size, or rehash-threshold, respectively, as used in ƒ**make-hash-table**.

(ƒ**sxhash** *foo*)
▷ Hash code unique for any argument ƒ**equal** *foo*.

8 Structures

(*m*defstruct
{*foo*
{ { **:conc-name**
| (**:conc-name** [*slot-prefix*[*foo*-]]) }
{ **:constructor**
| (**:constructor** [*maker*[MAKE-*foo*] [(*ord-λ**)]]) }*
{ **:copier**
| (**:copier** [*copier*[COPY-*foo*]]) }
(**:include** *struct* { *slot*
| (*slot* [*init* { **:type** *sl-type*
| **:read-only** *b* }]) }*)
{ (**:type** { **list**
| **vector**
| (**vector** *type*) })
{ **:named**
| (**:initial-offset** *n*) }
{ (**:print-object** [*o-printer*])
| (**:print-function** [*f-printer*]) }
{ **:predicate**
| (**:predicate** [*p-name*[*foo*-P]]) } } }
[*doc*] { *slot*
| (*slot* [*init* { **:type** *slot-type*
| **:read-only** *bool* }]) }*)
▷ Define structure *foo* together with functions `MAKE-`*foo*, `COPY-`*foo* and *foo*`-P`; and **setf**able accessors *foo*`-`*slot*. Instances are of class *foo* or, if **defstruct** option **:type** is given, of the specified type. They can be created by (`MAKE-`*foo* {**:***slot value*}*) or, if *ord-λ* (see page 17) is given, by (*maker arg** {**:***key value*}*). In the latter case, *arg*s and **:***key*s correspond to the positional and keyword parameters defined in *ord-λ* whose *var*s in turn correspond to *slot*s. **:print-object**/**:print-function** generate a *g***print-object** method for an instance *bar* of *foo* calling (*o-printer bar stream*) or (*f-printer bar stream print-level*), respectively. If **:type** without **:named** is given, no *foo*`-P` is created.

(ƒ**copy-structure** *structure*)
▷ Return copy of *structure* with shared slot values.

9 Control Structure

9.1 Predicates

(ƒ**eq** *foo bar*) ▷ `T` if *foo* and *bar* are identical.

(ƒ**eql** *foo bar*)
▷ `T` if *foo* and *bar* are identical, or the same **character**, or **number**s of the same type and value.

(ƒ**equal** *foo bar*)
▷ `T` if *foo* and *bar* are ƒ**eql**, or are equivalent **pathname**s, or are **cons**es with ƒ**equal** cars and cdrs, or are **string**s or **bit-vector**s with ƒ**eql** elements below their fill pointers.

(ƒ**equalp** *foo bar*)
▷ `T` if *foo* and *bar* are identical; or are the same **character** ignoring case; or are **number**s of the same value ignoring type; or are equivalent **pathname**s; or are **cons**es or **array**s of the same shape with ƒ**equalp** elements; or are structures of the same type with ƒ**equalp** elements; or are **hash-table**s of the same size with the same **:test** function, the same keys in terms of **:test** function, and ƒ**equalp** elements.

(ƒ**not** *foo*) ▷ `T` if *foo* is `NIL`; `NIL` otherwise.

(ƒ**boundp** *symbol*) ▷ `T` if *symbol* is a special variable.

(ƒ**constantp** *foo* [*environment*[NIL]])
▷ `T` if *foo* is a constant form.

(ƒ**functionp** *foo*) ▷ `T` if *foo* is of type **function**.

(${}_m$**ecase** | **ccase** *test* ((*key**) | *key*) *foo*P*)*)

▷ Return the values of the first *foo** one of whose *key*s is **eql** *test*. Signal non-correctable/correctable **type-error** if there is no matching *key*.

(${}_m$**and** *form**T)

▷ Evaluate *form*s from left to right. Immediately return NIL if one *form*'s value is NIL. Return values of last *form* otherwise.

(${}_m$**or** *form**NIL)

▷ Evaluate *form*s from left to right. Immediately return primary value of first non-NIL-evaluating form, or all values if last *form* is reached. Return NIL if no *form* returns T.

(${}_s$**progn** *form**NIL)

▷ Evaluate *form*s sequentially. Return values of last *form*.

(${}_s$**multiple-value-prog1** *form-r form**)
(${}_m$**prog1** *form-r form**)
(${}_m$**prog2** *form-a form-r form**)

▷ Evaluate forms in order. Return values/primary value, respectively, of *form-r*.

(${}_m$**prog** | **prog*** (*name* | (*name* [*value*NIL])*) (**declare** *decl**)* *tag* | *form**)

▷ Evaluate ${}_s$**tagbody**-like body with *name*s lexically bound (in parallel or sequentially, respectively) to *value*s. Return NIL or explicitly ${}_m$**return**ed values. Implicitly, the whole form is a ${}_s$**block** named NIL.

(${}_s$**unwind-protect** *protected cleanup**)

▷ Evaluate *protected* and then, no matter how control leaves *protected*, *cleanup*s. Return values of *protected*.

(${}_s$**block** *name form*P*)

▷ Evaluate *form*s in a lexical environment, and return their values unless interrupted by ${}_s$**return-from**.

(${}_s$**return-from** *foo* [*result*NIL])
(${}_m$**return** [*result*NIL])

▷ Have nearest enclosing ${}_s$**block** named *foo*/named NIL, respectively, return with values of *result*.

(${}_s$**tagbody** {*tag* | *form*}*)

▷ Evaluate *form*s in a lexical environment. *tag*s (symbols or integers) have lexical scope and dynamic extent, and are targets for ${}_s$**go**. Return NIL.

(${}_s$**go** *tag*)

▷ Within the innermost possible enclosing ${}_s$**tagbody**, jump to a tag ${}_f$**eql** *tag*.

(${}_s$**catch** *tag form*P*)

▷ Evaluate *form*s and return their values unless interrupted by ${}_s$**throw**.

(${}_s$**throw** *tag form*)

▷ Have the nearest dynamically enclosing ${}_s$**catch** with a tag ${}_f$**eq** *tag* return with the values of *form*.

(${}_f$**sleep** *n*) ▷ Wait *n* seconds; return NIL.

9.6 Iteration

(${}_m$**do** | **do*** (*var* | (*var* [*start* [*step*]])*) (*stop result*P*) (**declare** *decl**)* *tag* | *form**)

▷ Evaluate ${}_s$**tagbody**-like body with *var*s successively bound according to the values of the corresponding *start* and *step* forms. *var*s are bound in parallel/sequentially, respectively. Stop iteration when *stop* is T. Return values of *result**. Implicitly, the whole form is a ${}_s$**block** named NIL.

9.3 Functions

Below, ordinary lambda list (*ord-λ**) has the form
(*var** [**&optional** *var* | (*var* [*init*NIL [*supplied-p*]])*] [**&rest** *var*]
[**&key** *var* | ((:*key* *var*) | *var*) [*init*NIL [*supplied-p*]])* [**&allow-other-keys**]]
[**&aux** *var* | (*var* [*init*NIL])*]).

supplied-p is T if there is a corresponding argument. *init* forms can refer to any *init* and *supplied-p* to their left.

(${}_m$**defun** *foo* | (**setf** *foo*) | ${}_m$**lambda** (*ord-λ**) (**declare** *decl**)* | *doc* *form*P*)

▷ Define a function named *foo* or (**setf** *foo*), or an anonymous function, respectively, which applies *form*s to *ord-λ*s. For ${}_m$**defun**, *form*s are enclosed in an implicit ${}_s$**block** named *foo*.

(${}_s$**flet** | ${}_s$**labels** ((*foo* (*ord-λ**) | (**setf** *foo*) (*new-value ord-λ**)) (**declare** *local-decl**)* | *doc* *local-form*P*)*) (**declare** *decl**)* *form*P*)

▷ Evaluate *form*s with locally defined functions *foo*. Globally defined functions of the same name are shadowed. Each *foo* is also the name of an implicit ${}_s$**block** around its corresponding *local-form**. Only for ${}_s$**labels**, functions *foo* are visible inside *local-forms*. Return values of *form*s.

(${}_s$**function** *foo* | (${}_m$**lambda** *form**))

▷ Return lexically innermost function named *foo* or a lexical closure of the ${}_m$**lambda** expression.

(${}_f$**apply** *function* | (**setf** *function*) *arg** *args*)

▷ Values of *function* called with *arg*s and the list elements of *args*. **setf**able if *function* is one of ${}_f$**aref**, ${}_f$**bit**, and ${}_f$**sbit**.

(${}_f$**funcall** *function* arg*) ▷ Values of *function* called with *arg*s.

(${}_s$**multiple-value-call** *function form**)

▷ Call *function* with all the values of each *form* as its arguments. Return values returned by *function*.

(${}_f$**values-list** *list*) ▷ Return elements of *list*.

(${}_f$**values** *foo**)

▷ Return as multiple values the primary values of the *foo*s. **setf**able.

(${}_f$**multiple-value-list** *form*) ▷ List of the values of *form*.

(${}_m$**nth-value** *n form*)

▷ Zero-indexed *n*th return value of *form*.

(${}_f$**complement** *function*)

▷ Return new function with same arguments and same side effects as *function*, but with complementary truth value.

(${}_f$**constantly** *foo*)

▷ Function of any number of arguments returning *foo*.

(${}_f$**identity** *foo*) ▷ Return *foo*.

(${}_f$**function-lambda-expression** *function*)

▷ If available, return lambda expression of *function*, NIL if *function* was defined in an environment without bindings, and name of *function*.
2 3

(${}_f$**fdefinition** *foo* | (**setf** *foo*))

▷ Definition of global function *foo*. **setf**able.

(*f* **fmakunbound** *foo*)
▷ Remove global function or macro definition *foo*.

c **call-arguments-limit**
c **lambda-parameters-limit**
▷ Upper bound of the number of function arguments or lambda list parameters, respectively; ≥ 50.

c **multiple-values-limit**
▷ Upper bound of the number of values a multiple value can have; ≥ 20.

9.4 Macros

Below, macro lambda list (*macro-λ**) has the form of either

([**&whole** *var*] [*E*] { *var* | (*macro-λ**) }* [*E*]
[**&optional** { *var* | ({ *var* | (*macro-λ**) } [*init*NIL [*supplied-p*]]) }*] [*E*]
[{ **&rest** | **&body** } { *rest-var* | (*macro-λ**) }] [*E*]
[**&key** { *var* | ({ *var* | (**:***key* { *var* | (*macro-λ**) }) } [*init*NIL [*supplied-p*]]) }* [*E*]
[**&allow-other-keys**]] [**&aux** { *var* | (*var* [*init*NIL]) }*] [*E*])
or
([**&whole** *var*] [*E*] { *var* | (*macro-λ**) }* [*E*] [**&optional**
{ *var* | ({ *var* | (*macro-λ**) } [*init*NIL [*supplied-p*]]) }*] [*E*] **.** *rest-var*).

One toplevel [*E*] may be replaced by **&environment** *var*. *supplied-p* is `T` if there is a corresponding argument. *init* forms can refer to any *init* and *supplied-p* to their left.

({ *m* **defmacro** | *f* **define-compiler-macro** } { *foo* | (**setf** *foo*) } (*macro-λ**)
{ | (**declare** *decl**)* | *doc* } *form*P*)
▷ Define macro *foo* which on evaluation as (*foo tree*) applies expanded *form*s to arguments from *tree*, which corresponds to *tree*-shaped *macro-λ*s. *form*s are enclosed in an implicit *s* **block** named *foo*.

(*m* **define-symbol-macro** *foo form*)
▷ Define symbol macro *foo* which on evaluation evaluates expanded *form*.

(*s* **macrolet** ((*foo* (*macro-λ**) { | (**declare** *local-decl**)* | *doc* }
*macro-form*P*)*) (**declare** *decl**)* *form*P*)
▷ Evaluate *forms* with locally defined mutually invisible macros *foo* which are enclosed in implicit *s* **block**s of the same name.

(*s* **symbol-macrolet** ((*foo expansion-form*)*) (**declare** *decl**)* *form*P*)
▷ Evaluate *forms* with locally defined symbol macros *foo*.

(*m* **defsetf** *function* { *updater* [*doc*] | (*setf-λ**) (*s-var**) { | (**declare** *decl**)* | *doc* } *form*P* })
where defsetf lambda list (*setf-λ**) has the form
(*var** [**&optional** { *var* | (*var* [*init*NIL [*supplied-p*]]) }*] [**&rest** *var*]
[**&key** { *var* | ({ *var* | (**:***key var*) } [*init*NIL [*supplied-p*]]) }*
[**&allow-other-keys**]] [**&environment** *var*])
▷ Specify how to **setf** a place accessed by *function*. **Short form:** (**setf** (*function arg**) *value-form*) is replaced by (*updater arg* value-form*); the latter must return *value-form*. **Long form:** on invocation of (**setf** (*function arg**) *value-form*), *form*s must expand into code that sets the place accessed where *setf-λ* and *s-var** describe the arguments of *function* and the value(s) to be stored, respectively; and that returns the value(s) of *s-var**. *form*s are enclosed in an implicit *s* **block** named *function*.

(*m* **define-setf-expander** *function* (*macro-λ**) { | (**declare** *decl**)* | *doc* }
*form*P*)
▷ Specify how to **setf** a place accessed by *function*. On invocation of (**setf** (*function arg**) *value-form*), *form** must expand into code returning *arg-vars*, *args*, *newval-vars*, *set-form*, and *get-form* as described with *f* **get-setf-expansion** where the elements of macro lambda list *macro-λ** are bound to corresponding *arg*s. *form*s are enclosed in an implicit *s* **block** named *function*.

(*f* **get-setf-expansion** *place* [*environment*NIL])
▷ Return lists of temporary variables *arg-vars* and of corresponding *args*[2] as given with *place*, list *newval-vars*[3] with temporary variables corresponding to the new values, and *set-form*[4] and *get-form*[5] specifying in terms of *arg-vars* and *newval-vars* how to **setf** and how to read *place*.

(*m* **define-modify-macro** *foo* ([**&optional**
{ *var* | (*var* [*init*NIL [*supplied-p*]]) }*] [**&rest** *var*]) *function* [*doc*])
▷ Define macro *foo* able to modify a place. On invocation of (*foo place arg**), the value of *function* applied to *place* and *arg*s will be stored into *place* and returned.

c **lambda-list-keywords**
▷ List of macro lambda list keywords. These are at least:

&whole *var*
▷ Bind *var* to the entire macro call form.

&optional *var**
▷ Bind *var*s to corresponding arguments if any.

{**&rest**|**&body**} *var*
▷ Bind *var* to a list of remaining arguments.

&key *var**
▷ Bind *var*s to corresponding keyword arguments.

&allow-other-keys
▷ Suppress keyword argument checking. Callers can do so using **:allow-other-keys** `T`.

&environment *var*
▷ Bind *var* to the lexical compilation environment.

&aux *var** ▷ Bind *var*s as in *s* **let***.

9.5 Control Flow

(*s* **if** *test then* [*else*NIL])
▷ Return values of *then* if *test* returns `T`; return values of *else* otherwise.

(*m* **cond** (*test then*P*[*test*])*)
▷ Return the values of the first *then** whose *test* returns `T`; return `NIL` if all *test*s return `NIL`.

({ *m* **when** | *m* **unless** } *test foo*P*)
▷ Evaluate *foo*s and return their values if *test* returns `T` or `NIL`, respectively. Return `NIL` otherwise.

(*m* **case** *test* ({ (*key**) | *key* } *foo*P*)* [({ **otherwise** | `T` } *bar*P*)NIL])
▷ Return the values of the first *foo** one of whose *key*s is **eql** *test*. Return values of *bar*s if there is no matching *key*.

(slot
  ((slot { :reader reader }*
          { :writer { writer | (setf writer) } }*
          { :accessor accessor }*
          :allocation { :instance | :class } [:instance]
          { :initarg :initarg-name }*
          :initform form
          :type type
          :documentation slot-doc))*)
{ (:default-initargs { name value }*) | (:documentation class-doc) | (:metaclass name [standard-class]) })

▷ Define or modify class *foo* as a subclass of *superclass*es. Transform existing instances, if any, by ₘ**make-instances-obsolete**. In a new instance *i* of *foo*, a *slot*'s value defaults to *form* unless set via :*initarg-name*; it is readable via (*reader i*) or (*accessor i*), and writable via (*writer value i*) or (**setf** (*accessor i*) *value*). *slot*s with **:allocation :class** are shared by all instances of class *foo*.

(f **find-class** *symbol* [*errorp*[T] [*environment*]])
▷ Return class named *symbol*. **setf**able.

(g **make-instance** *class* {:*initarg value*}* *other-keyarg**)
▷ Make new instance of *class*.

(g **reinitialize-instance** *instance* {:*initarg value*}* *other-keyarg**)
▷ Change local slots of *instance* according to *initarg*s by means of g**shared-initialize**.

(f **slot-value** *foo slot*) ▷ Return value of *slot* in *foo*. **setf**able.

(f **slot-makunbound** *instance slot*)
▷ Make *slot* in *instance* unbound.

({ m**with-slots** ({ *slot* | (*var slot*) }*)
  | m**with-accessors** ((*var accessor*)*) } *instance* (**declare** *decl**)*
form[P]*)
▷ Return values of *forms* after evaluating them in a lexical environment with slots of *instance* visible as **setf**able *slot*s or *var*s/with *accessor*s of *instance* visible as **setf**able *var*s.

(g **class-name** *class*)
((**setf** g**class-name**) *new-name class*) ▷ Get/set name of *class*.

(f **class-of** *foo*) ▷ Class *foo* is a direct instance of.

(g **change-class** *instance new-class* {:*initarg value*}* *other-keyarg**)
▷ Change class of *instance* to *new-class*. Retain the status of any slots that are common between *instance*'s original class and *new-class*. Initialize any newly added slots with the *value*s of the corresponding *initarg*s if any, or with the values of their **:initform** forms if not.

(g **make-instances-obsolete** *class*)
▷ Update all existing instances of *class* using g**update-instance-for-redefined-class**.

({ g**initialize-instance** *instance*
  | g**update-instance-for-different-class** *previous current* }
{:*initarg value*}* *other-keyarg**)
▷ Set slots on behalf of g**make-instance**/of g**change-class** by means of g**shared-initialize**.

(g **update-instance-for-redefined-class** *new-instance added-slots discarded-slots discarded-slots-property-list* {:*initarg value*}* *other-keyarg**)
▷ On behalf of g**make-instances-obsolete** and by means of g**shared-initialize**, set any *initarg* slots to their corresponding *value*s; set any remaining *added-slots* to the values of their **:initform** forms. Not to be called by user.

(g **allocate-instance** *class* {:*initarg value*}* *other-keyarg**)
▷ Return uninitialized instance of *class*. Called by g**make-instance**.

(m **dotimes** (*var i* [*result*[NIL]]) (**declare** *decl**)* {*tag* | *form*}*)
▷ Evaluate s**tagbody**-like body with *var* successively bound to integers from 0 to $i-1$. Upon evaluation of *result*, *var* is *i*. Implicitly, the whole form is a s**block** named NIL.

(m **dolist** (*var list* [*result*[NIL]]) (**declare** *decl**)* {*tag* | *form*}*)
▷ Evaluate s**tagbody**-like body with *var* successively bound to the elements of *list*. Upon evaluation of *result*, *var* is NIL. Implicitly, the whole form is a s**block** named NIL.

9.7 Loop Facility

(m **loop** *form**)
▷ **Simple Loop.** If *form*s do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit s**block** named NIL.

(m **loop** *clause**)
▷ **Loop Facility.** For Loop Facility keywords see below and Figure 1.

named *n*[NIL] ▷ Give m**loop**'s implicit s**block** a name.

{**with** { *var-s* | (*var-s**) } [*d-type*] [= *foo*]}+
{**and** { *var-p* | (*var-p**) } [*d-type*] [= *bar*]}*
where destructuring type specifier *d-type* has the form
{ **fixnum** | **float** | T | NIL | {**of-type** { *type* | (*type**) }} }
▷ Initialize (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel.

{{**for** | **as**} { *var-s* | (*var-s**) } [*d-type*]}+ {**and** { *var-p* | (*var-p**) } [*d-type*]}*
▷ Begin of iteration control clauses. Initialize and step (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel. Destructuring type specifier *d-type* as with **with**.

{**upfrom** | **from** | **downfrom**} *start*
▷ Start stepping with *start*

{**upto** | **downto** | **to** | **below** | **above**} *form*
▷ Specify *form* as the end value for stepping.

{**in** | **on**} *list*
▷ Bind *var* to successive elements/tails, respectively, of *list*.

by {*step*[1] | *function*[#'cdr]}
▷ Specify the (positive) decrement or increment or the *function* of one argument returning the next part of the list.

= *foo* [**then** *bar*[*foo*]]
▷ Bind *var* initially to *foo* and later to *bar*.

across *vector*
▷ Bind *var* to successive elements of *vector*.

being {**the** | **each**}
▷ Iterate over a hash table or a package.

{**hash-key** | **hash-keys**} {**of** | **in**} *hash-table* [**using** (**hash-value** *value*)]
▷ Bind *var* successively to the keys of *hash-table*; bind *value* to corresponding values.

{**hash-value** | **hash-values**} {**of** | **in**} *hash-table* [**using** (**hash-key** *key*)]
▷ Bind *var* successively to the values of *hash-table*; bind *key* to corresponding keys.

{**symbol** | **symbols** | **present-symbol** | **present-symbols** | **external-symbol** | **external-symbols**} [{**of** | **in**} *package*[v*package**]]
▷ Bind *var* successively to the accessible symbols, or the present symbols, or the external symbols respectively, of *package*.

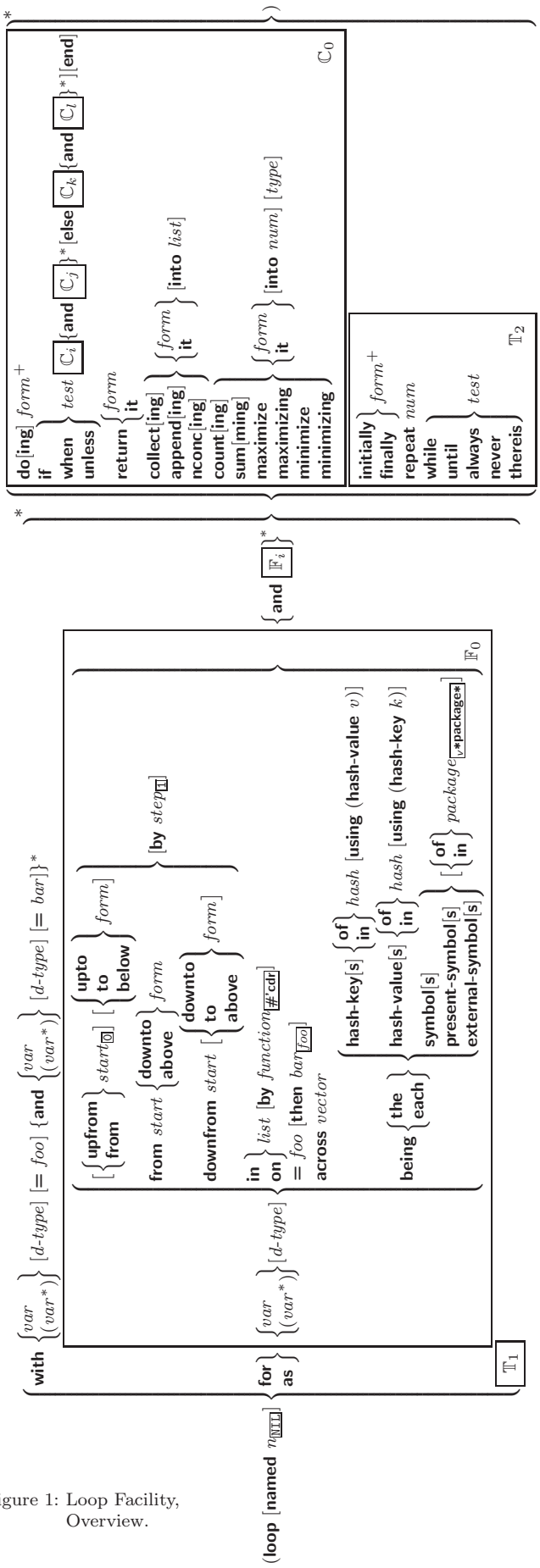

Figure 1: Loop Facility, Overview.

{**do**|**doing**} *form*$^+$
▷ Evaluate *form*s in every iteration.

{**if**|**when**|**unless**} *test i-clause* {**and** *j-clause*}* [**else** *k-clause* {**and** *l-clause*}*] [**end**]
▷ If *test* returns `T`, `T`, or `NIL`, respectively, evaluate *i-clause* and *j-clause*s; otherwise, evaluate *k-clause* and *l-clause*s.

it ▷ Inside *i-clause* or *k-clause*: value of *test*.

return {*form*|**it**}
▷ Return immediately, skipping any **finally** parts, with values of *form* or **it**.

{**collect**|**collecting**} {*form*|**it**} [**into** *list*]
▷ Collect values of *form* or **it** into *list*. If no *list* is given, collect into an anonymous list which is returned after termination.

{**append**|**appending**|**nconc**|**nconcing**} {*form*|**it**} [**into** *list*]
▷ Concatenate values of *form* or **it**, which should be lists, into *list* by the means of $_f$**append** or $_f$**nconc**, respectively. If no *list* is given, collect into an anonymous list which is returned after termination.

{**count**|**counting**} {*form*|**it**} [**into** *n*] [*type*]
▷ Count the number of times the value of *form* or of **it** is `T`. If no *n* is given, count into an anonymous variable which is returned after termination.

{**sum**|**summing**} {*form*|**it**} [**into** *sum*] [*type*]
▷ Calculate the sum of the primary values of *form* or of **it**. If no *sum* is given, sum into an anonymous variable which is returned after termination.

{**maximize**|**maximizing**|**minimize**|**minimizing**} {*form*|**it**} [**into** *max-min*] [*type*]
▷ Determine the maximum or minimum, respectively, of the primary values of *form* or of **it**. If no *max-min* is given, use an anonymous variable which is returned after termination.

{**initially**|**finally**} *form*$^+$
▷ Evaluate *form*s before begin, or after end, respectively, of iterations.

repeat *num*
▷ Terminate $_m$**loop** after *num* iterations; *num* is evaluated once.

{**while**|**until**} *test*
▷ Continue iteration until *test* returns `NIL` or `T`, respectively.

{**always**|**never**} *test*
▷ Terminate $_m$**loop** returning `NIL` and skipping any **finally** parts as soon as *test* is `NIL` or `T`, respectively. Otherwise continue $_m$**loop** with its default return value set to `T`.

thereis *test*
▷ Terminate $_m$**loop** when *test* is `T` and return value of *test*, skipping any **finally** parts. Otherwise continue $_m$**loop** with its default return value set to `NIL`.

($_m$**loop-finish**)
▷ Terminate $_m$**loop** immediately executing any **finally** clauses and returning any accumulated results.

10 CLOS

10.1 Classes

($_f$**slot-exists-p** *foo bar*) ▷ `T` if *foo* has a slot *bar*.

($_f$**slot-boundp** *instance slot*) ▷ `T` if *slot* in *instance* is bound.

($_m$**defclass** *foo* (*superclass** **standard-object**)

($_f$**cerror** *continue-control* {*condition continue-arg** | *condition-type* {**:***initarg-name value*}* | *control arg**})

▷ Unless handled, signal as correctable **error** *condition* or a new instance of *condition-type* or, with $_f$**format** *control* and *arg*s (see page 36), **simple-error**. In the debugger, use $_f$**format** arguments *continue-control* and *continue-arg*s to tag the continue option. Return NIL.

($_m$**ignore-errors** *form*^{P*})

▷ Return values of *forms* or, in case of **error**s, NIL and the condition.

($_f$**invoke-debugger** *condition*)

▷ Invoke debugger with *condition*.

($_m$**assert** *test* [(*place**) [{*condition continue-arg** | *condition-type* {**:***initarg-name value*}* | *control arg**}]])

▷ If *test*, which may depend on *place*s, returns NIL, signal as correctable **error** *condition* or a new instance of *condition-type* or, with $_f$**format** *control* and *arg*s (see page 36), **error**. When using the debugger's continue option, *place*s can be altered before re-evaluation of *test*. Return NIL.

($_m$**handler-case** *foo* (*type* ([*var*]) (**declare** $\widehat{\textit{decl}^*}$)* *condition-form*^{P*})* [(**:no-error** (*ord-λ**) (**declare** $\widehat{\textit{decl}^*}$)* *form*^{P*})])

▷ If, on evaluation of *foo*, a condition of *type* is signalled, evaluate matching *condition-form*s with *var* bound to the condition, and return their values. Without a condition, bind *ord-λ*s to values of *foo* and return values of *forms* or, without a **:no-error** clause, return values of *foo*. See page 17 for (*ord-λ**).

($_m$**handler-bind** ((*condition-type handler-function*)*) *form*^{P*})

▷ Return values of *forms* after evaluating them with *condition-type*s dynamically bound to their respective *handler-function*s of argument condition.

($_m$**with-simple-restart** ({*restart* | NIL} *control arg**) *form*^{P*})

▷ Return values of *form*s unless *restart* is called during their evaluation. In this case, describe *restart* using $_f$**format** *control* and *arg*s (see page 36) and return NIL and T.

($_m$**restart-case** *form* (*restart* (*ord-λ**) {**:interactive** *arg-function* | **:report** {*report-function* | *string*["*restart*"]} | **:test** *test-function*[T]}

(**declare** $\widehat{\textit{decl}^*}$)* *restart-form*^{P*})*)

▷ Return values of *form* or, if during evaluation of *form* one of the dynamically established *restart*s is called, the values of its *restart-form*s. A *restart* is visible under *condition* if (**funcall #'***test-function condition*) returns T. If presented in the debugger, *restart*s are described by *string* or by **#'***report-function* (of a stream). A *restart* can be called by (**invoke-restart** *restart arg**), where *arg*s match *ord-λ**, or by (**invoke-restart-interactively** *restart*) where a list of the respective *arg*s is supplied by **#'***arg-function*. See page 17 for *ord-λ**.

($_m$**restart-bind** (({$\widehat{\textit{restart}}$ | NIL} *restart-function*

{**:interactive-function** *arg-function* | **:report-function** *report-function* | **:test-function** *test-function*})*) *form*^{P*})

▷ Return values of *forms* evaluated with dynamically established *restart*s whose *restart-function*s should perform a non-local transfer of control. A restart is visible under *condition* if (*test-function condition*) returns T. If presented in the debugger, *restart*s are described by *restart-function* (of a stream). A *restart* can be called by (**invoke-restart** *restart arg**), where *arg*s must be suitable for the corresponding *restart-function*, or by (**invoke-restart-interactively** *restart*) where a list of the respective *arg*s is supplied by *arg-function*.

($_g$**shared-initialize** *instance* {*initform-slots* | T} {**:***initarg-slot value*}* *other-keyarg**)

▷ Fill the *initarg-slot*s of *instance* with the corresponding *value*s, and fill those *initform-slots* that are not *initarg-slot*s with the values of their **:initform** forms.

($_g$**slot-missing** *class instance slot* {**setf** | **slot-boundp** | **slot-makunbound** | **slot-value**} [*value*])

($_g$**slot-unbound** *class instance slot*)

▷ Called on attempted access to non-existing or unbound *slot*. Default methods signal **error**/**unbound-slot**, respectively. Not to be called by user.

10.2 Generic Functions

($_f$**next-method-p**) ▷ T if enclosing method has a next method.

($_m$**defgeneric** {*foo* | (**setf** *foo*)} (*required-var** [**&optional** {*var* | (*var*)}*] [**&rest** *var*] [**&key** {*var* | (*var* | (**:***key var*))}* [**&allow-other-keys**]])

{(**:argument-precedence-order** *required-var*$^+$) | (**declare** (**optimize** *method-selection-optimization*)$^+$) | (**:documentation** $\widehat{\textit{string}}$) | (**:generic-function-class** *gf-class*[standard-generic-function]) | (**:method-class** *method-class*[standard-method]) | (**:method-combination** *c-type*[standard] *c-arg**) | (**:method** *defmethod-args*)*}

▷ Define or modify generic function *foo*. Remove any methods previously defined by defgeneric. *gf-class* and the lambda paramters *required-var** and *var** must be compatible with existing methods. *defmethod-args* resemble those of $_m$**defmethod**. For *c-type* see section 10.3.

($_f$**ensure-generic-function** {*foo* | (**setf** *foo*)}

{**:argument-precedence-order** *required-var*$^+$ | **:declare** (**optimize** *method-selection-optimization*) | **:documentation** *string* | **:generic-function-class** *gf-class* | **:method-class** *method-class* | **:method-combination** *c-type c-arg** | **:lambda-list** *lambda-list* | **:environment** *environment*})

▷ Define or modify generic function *foo*. *gf-class* and *lambda-list* must be compatible with a pre-existing generic function or with existing methods, respectively. Changes to *method-class* do not propagate to existing methods. For *c-type* see section 10.3.

($_m$**defmethod** {*foo* | (**setf** *foo*)} [{**:before** | **:after** | **:around** | *qualifier**}] [primary method]

({*var* | (*spec-var* {*class* | (**eql** *bar*)})}* [**&optional** {*var* | (*var* [*init* [*supplied-p*]])}*] [**&rest** *var*] [**&key** {*var* | ({*var* | (**:***key var*)} [*init* [*supplied-p*]])}* [**&allow-other-keys**]] [**&aux** {*var* | (*var* [*init*])}*]) {(**declare** $\widehat{\textit{decl}^*}$)* | $\widehat{\textit{doc}}$} *form*^{P*})

▷ Define new method for generic function *foo*. *spec-var*s specialize to either being of *class* or being **eql** *bar*, respectively. On invocation, *var*s and *spec-var*s of the new method act like parameters of a function with body *form**. *form*s are enclosed in an implicit $_s$**block** *foo*. Applicable *qualifier*s depend on the **method-combination** type; see section 10.3.

(g{**add-method** | **remove-method**} *generic-function method*)
▷ Add (if necessary) or remove (if any) *method* to/from *generic-function*.

(g**find-method** *generic-function qualifiers specializers* [*error*T])
▷ Return suitable method, or signal **error**.

(g**compute-applicable-methods** *generic-function args*)
▷ List of methods suitable for *args*, most specific first.

(f**call-next-method** *arg**current args)
▷ From within a method, call next method with *arg*s; return its values.

(g**no-applicable-method** *generic-function arg**)
▷ Called on invocation of *generic-function* on *arg*s if there is no applicable method. Default method signals **error**. Not to be called by user.

(f{**invalid-method-error** *method* | **method-combination-error**} *control arg**)
▷ Signal **error** on applicable method with invalid qualifiers, or on method combination. For *control* and *arg*s see **format**, page 36.

(g**no-next-method** *generic-function method arg**)
▷ Called on invocation of **call-next-method** when there is no next method. Default method signals **error**. Not to be called by user.

(g**function-keywords** *method*)
▷ Return list of keyword parameters of *method* and T2 if other keys are allowed.

(g**method-qualifiers** *method*) ▷ List of qualifiers of *method*.

10.3 Method Combination Types

standard
▷ Evaluate most specific **:around** method supplying the values of the generic function. From within this method, f**call-next-method** can call less specific **:around** methods if there are any. If not, or if there are no **:around** methods at all, call all **:before** methods, most specific first, and the most specific primary method which supplies the values of the calling f**call-next-method** if any, or of the generic function; and which can call less specific primary methods via f**call-next-method**. After its return, call all **:after** methods, least specific first.

and|**or**|**append**|**list**|**nconc**|**progn**|**max**|**min**|**+**
▷ Simple built-in **method-combination** types; have the same usage as the *c-type*s defined by the short form of m**define-method-combination**.

(m**define-method-combination** *c-type*
{**:documentation** *string* | **:identity-with-one-argument** *bool*NIL | **:operator** *operator*c-type})
▷ **Short Form.** Define new **method-combination** *c-type*. In a generic function using *c-type*, evaluate most specific **:around** method supplying the values of the generic function. From within this method, f**call-next-method** can call less specific **:around** methods if there are any. If not, or if there are no **:around** methods at all, return from the calling **call-next-method** or from the generic function, respectively, the values of (*operator* (*primary-method gen-arg**)*), *gen-arg** being the arguments of the generic function. The *primary-method*s are ordered [{**:most-specific-first** | **:most-specific-last**}:most-specific-first] (specified as *c-arg* in m**defgeneric**). Using *c-type* as the *qualifier* in m**defmethod** makes the method primary.

(m**define-method-combination** *c-type* (*ord-λ**) ((*group*
{***** | (*qualifier** [*****]) | *predicate*}
{**:description** *control* | **:order** {**:most-specific-first** | **:most-specific-last**}:most-specific-first | **:required** *bool*}*)
{(**:arguments** *method-combination-λ**) | (**:generic-function** *symbol*) | {(**declare** *decl**)* | *doc*} } *body*P*)
▷ **Long Form.** Define new **method-combination** *c-type*. A call to a generic function using *c-type* will be equivalent to a call to the forms returned by *body** with *ord-λ** bound to *c-arg** (cf. m**defgeneric**), with *symbol* bound to the generic function, with *method-combination-λ** bound to the arguments of the generic function, and with *group*s bound to lists of methods. An applicable method becomes a member of the leftmost *group* whose *predicate* or *qualifier*s match. Methods can be called via m**call-method**. Lambda lists (*ord-λ**) and (*method-combination-λ**) according to *ord-λ* on page 17, the latter enhanced by an optional **&whole** argument.

(m**call-method**
{*method* | (m**make-method** *form*)} [({*next-method* | (m**make-method** *form*)}*)])
▷ From within an effective method form, call *method* with the arguments of the generic function and with information about its *next-method*s; return its values.

11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 30.

(m**define-condition** *foo* (*parent-type**condition)
({*slot* | (*slot* {**:reader** *reader*}* {**:writer** {*writer* | (**setf** *writer*)}}* {**:accessor** *accessor*}* **:allocation** {**:instance** | **:class**}:instance {**:initarg** *:initarg-name*}* **:initform** *form* **:type** *type* **:documentation** *slot-doc*)*
{(**:default-initargs** {*name value*}*) | (**:documentation** *condition-doc*) | (**:report** {*string* | *report-function*})})
▷ Define, as a subtype of *parent-type*s, condition type *foo*. In a new condition, a *slot*'s value defaults to *form* unless set via **:***initarg-name*; it is readable via (*reader i*) or (*accessor i*), and writable via (*writer value i*) or (**setf** (*accessor i*) *value*). With **:allocation :class**, *slot* is shared by all conditions of type *foo*. A condition is reported by *string* or by *report-function* of arguments condition and stream.

(f**make-condition** *condition-type* {**:***initarg-name value*}*)
▷ Return new instance of *condition-type*.

({f**signal** | f**warn** | f**error**} {*condition* | *condition-type* {**:***initarg-name value*}* | *control arg**})
▷ Unless handled, signal as **condition**, **warning** or **error**, respectively, *condition* or a new instance of *condition-type* or, with f**format** *control* and *arg*s (see page 36), **simple-condition**, **simple-warning**, or **simple-error**, respectively. From f**signal** and f**warn**, return NIL.

13.2 Reader

(f{**y-or-n-p** **yes-or-no-p**} [*control* *arg**])
▷ Ask user a question and return T or NIL depending on their answer. See page 36, f**format**, for *control* and *args*.

(m**with-standard-io-syntax** *form*P*)
▷ Evaluate *form*s with standard behaviour of reader and printer. Return values of *form*s.

(f{**read** **read-preserving-whitespace**} [*stream*v**standard-input**] [*eof-err*T [*eof-val*NIL [*recursive*NIL]]])
▷ Read printed representation of object.

(f**read-from-string** *string* [*eof-error*T [*eof-val*NIL [{**:start** *start*0 **:end** *end*NIL **:preserve-whitespace** *bool*NIL}]]])
▷ Return object read from string and zero-indexed position$_2$ of next character.

(f**read-delimited-list** *char* [*stream*v**standard-input**] [*recursive*NIL])
▷ Continue reading until encountering *char*. Return list of objects read. Signal error if no *char* is found in stream.

(f**read-char** [*stream*v**standard-input**] [*eof-err*T [*eof-val*NIL [*recursive*NIL]]])
▷ Return next character from *stream*.

(f**read-char-no-hang** [*stream*v**standard-input**] [*eof-error*T [*eof-val*NIL [*recursive*NIL]]])
▷ Next character from *stream* or NIL if none is available.

(f**peek-char** [*mode*NIL [*stream*v**standard-input**] [*eof-error*T [*eof-val*NIL [*recursive*NIL]]]])
▷ Next, or if *mode* is T, next non-whitespace character, or if *mode* is a character, next instance of it, from *stream* without removing it there.

(f**unread-char** *character* [*stream*v**standard-input**])
▷ Put last f**read-char**ed *character* back into *stream*; return NIL.

(f**read-byte** *stream* [*eof-err*T [*eof-val*NIL]])
▷ Read next byte from binary *stream*.

(f**read-line** [*stream*v**standard-input**] [*eof-err*T [*eof-val*NIL [*recursive*NIL]]])
▷ Return a line of text from *stream* and T$_2$ if line has been ended by end of file.

(f**read-sequence** *sequence* *stream* [**:start** *start*0][**:end** *end*NIL])
▷ Replace elements of *sequence* between *start* and *end* with elements from binary or character *stream*. Return index of *sequence*'s first unmodified element.

(f**readtable-case** *readtable*)[**:upcase**]
▷ Case sensitivity attribute (one of **:upcase**, **:downcase**, **:preserve**, **:invert**) of *readtable*. **setf**able.

(f**copy-readtable** [*from-readtable*v**readtable**] [*to-readtable*NIL])
▷ Return copy of *from-readtable*.

(f**set-syntax-from-char** *to-char* *from-char* [*to-readtable*v**readtable**] [*from-readtable*standard readtable])
▷ Copy syntax of *from-char* to *to-readtable*. Return T.

v***readtable*** ▷ Current readtable.

v***read-base***10 ▷ Radix for reading **integer**s and **ratio**s.

v***read-default-float-format***single-float
▷ Floating point format to use when not indicated in the number read.

(f**invoke-restart** *restart* *arg**)
(f**invoke-restart-interactively** *restart*)
▷ Call function associated with *restart* with arguments given or prompted for, respectively. If *restart* function returns, return its values.

(f{**find-restart** **compute-restarts** *name*} [*condition*])
▷ Return innermost restart *name*, or a list of all restarts, respectively, out of those either associated with *condition* or un-associated at all; or, without *condition*, out of all restarts. Return NIL if search is unsuccessful.

(f**restart-name** *restart*) ▷ Name of *restart*.

(f{**abort** **muffle-warning** **continue** **store-value** *value* **use-value** *value*} [*condition*NIL])
▷ Transfer control to innermost applicable restart with same name (i.e. **abort**, ..., **continue** ...) out of those either associated with *condition* or un-associated at all; or, without *condition*, out of all restarts. If no restart is found, signal **control-error** for f**abort** and f**muffle-warning**, or return NIL for the rest.

(m**with-condition-restarts** *condition* *restarts* *form*P*)
▷ Evaluate *form*s with *restarts* dynamically associated with *condition*. Return values of *form*s.

(f**arithmetic-error-operation** *condition*)
(f**arithmetic-error-operands** *condition*)
▷ List of function or of its operands respectively, used in the operation which caused *condition*.

(f**cell-error-name** *condition*)
▷ Name of cell which caused *condition*.

(f**unbound-slot-instance** *condition*)
▷ Instance with unbound slot which caused *condition*.

(f**print-not-readable-object** *condition*)
▷ The object not readably printable under *condition*.

(f**package-error-package** *condition*)
(f**file-error-pathname** *condition*)
(f**stream-error-stream** *condition*)
▷ Package, path, or stream, respectively, which caused the *condition* of indicated type.

(f**type-error-datum** *condition*)
(f**type-error-expected-type** *condition*)
▷ Object which caused *condition* of type **type-error**, or its expected type, respectively.

(f**simple-condition-format-control** *condition*)
(f**simple-condition-format-arguments** *condition*)
▷ Return f**format** control or list of f**format** arguments, respectively, of *condition*.

v***break-on-signals***NIL
▷ Condition type debugger is to be invoked on.

v***debugger-hook***NIL
▷ Function of condition and function itself. Called before debugger.

12 Types and Classes

For any class, there is always a corresponding type of the same name.

(f**typep** *foo* *type* [*environment*NIL]) ▷ T if *foo* is of *type*.

(f**subtypep** *type-a* *type-b* [*environment*])
▷ Return T if *type-a* is a recognizable subtype of *type-b*, and NIL if the relationship could not be determined.$_2$

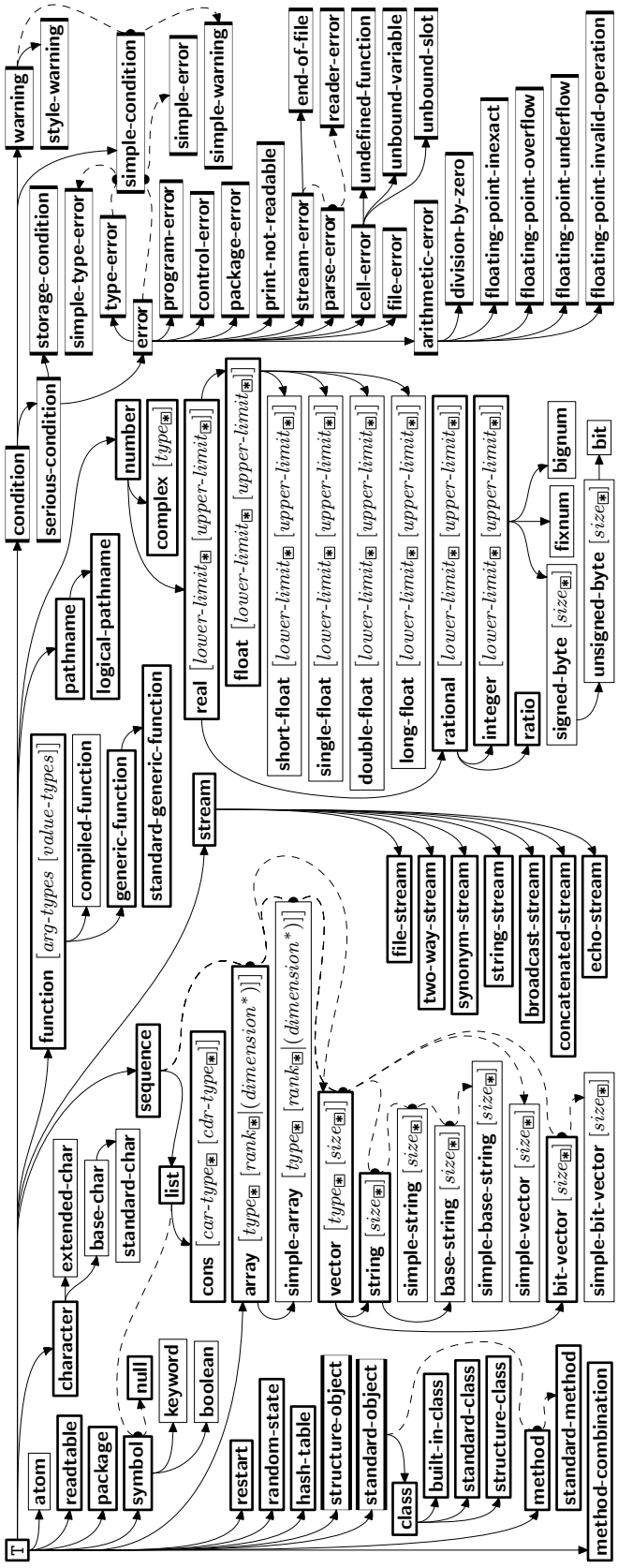

Figure 2: Precedence Order of System Classes (▭), Classes (▬), Types (▭), and Condition Types (▭).
Every type is also a supertype of NIL, the empty type.

(*s*the $\widehat{type}$ *form*) ▷ Declare values of *form* to be of *type*.

(*f*coerce *object type*) ▷ Coerce *object* into *type*.

(*m*typecase *foo* ($\widehat{type}$ *a-form*^{P_*})* [({otherwise | T} *b-form*$_{\texttt{NIL}}^{P_*}$)])
▷ Return values of the first *a-form** whose *type* is *foo* of. Return values of *b-forms* if no *type* matches.

({*m*etypecase | *m*ctypecase} *foo* ($\widehat{type}$ *form*^{P_*})*)
▷ Return values of the first *form** whose *type* is *foo* of. Signal non-correctable/correctable **type-error** if no *type* matches.

(*f*type-of *foo*) ▷ Type of *foo*.

(*m*check-type *place type* [*string*$_{\{\texttt{a}|\texttt{an}\}\ type}$])
▷ Signal correctable **type-error** if *place* is not of *type*. Return NIL.

(*f*stream-element-type *stream*) ▷ Type of *stream* objects.

(*f*array-element-type *array*) ▷ Element type *array* can hold.

(*f*upgraded-array-element-type *type* [*environment*$_{\texttt{NIL}}$])
▷ Element type of most specialized array capable of holding elements of *type*.

(*m*deftype *foo* (*macro-λ**) {(declare $\widehat{decl}$*)* | $\widehat{doc}$} *form*^{P_*})
▷ Define type *foo* which when referenced as (*foo* $\widehat{arg}$*) (or as *foo* if *macro-λ* doesn't contain any required parameters) applies expanded *form*s to *arg*s returning the new type. For (*macro-λ**) see page 18 but with default value of * instead of NIL. *form*s are enclosed in an implicit *s*block named *foo*.

(eql *foo*)
(member *foo**) ▷ Specifier for a type comprising *foo* or *foo*s.

(satisfies *predicate*)
▷ Type specifier for all objects satisfying *predicate*.

(mod *n*) ▷ Type specifier for all non-negative integers $< n$.

(not *type*) ▷ Complement of type.

(and *type*$^*_{\texttt{T}}$) ▷ Type specifier for intersection of *type*s.

(or *type*$^*_{\texttt{NIL}}$) ▷ Type specifier for union of *type*s.

(values *type** [&optional *type** [&rest *other-args*]])
▷ Type specifier for multiple values.

* ▷ As a type argument (cf. Figure 2): no restriction.

13 Input/Output

13.1 Predicates

(*f*streamp *foo*)
(*f*pathnamep *foo*) ▷ T if *foo* is of indicated type.
(*f*readtablep *foo*)

(*f*input-stream-p *stream*)
(*f*output-stream-p *stream*)
(*f*interactive-stream-p *stream*)
(*f*open-stream-p *stream*)
▷ Return T if *stream* is for input, for output, interactive, or open, respectively.

(*f*pathname-match-p *path wildcard*)
▷ T if *path* matches *wildcard*.

(*f*wild-pathname-p *path* [{:host|:device|:directory|:name|:type|:version|NIL}])
▷ Return T if indicated component in *path* is wildcard. (NIL indicates any component.)

*v****print-circle***[NIL]
▷ If T, avoid indefinite recursion while printing circular structure.

*v****print-escape***[T]
▷ If NIL, do not print escape characters and package prefixes.

*v****print-gensym***[T] ▷ If T, print **#:** before uninterned symbols.

*v****print-length***[NIL]
*v****print-level***[NIL]
*v****print-lines***[NIL]
▷ If integer, restrict printing of objects to that number of elements per level/to that depth/to that number of lines.

*v****print-miser-width***
▷ If integer and greater than the width available for printing a substructure, switch to the more compact miser style.

*v****print-pretty*** ▷ If T, print prettily.

*v****print-radix***[NIL] ▷ If T, print rationals with a radix indicator.

*v****print-readably***[NIL]
▷ If T, print *f***read**ably or signal error **print-not-readable**.

*v****print-right-margin***[NIL]
▷ Right margin width in ems while pretty-printing.

(*f***set-pprint-dispatch** *type function* [*priority*[0] [*table*[*v****print-pprint-dispatch***]])
▷ Install entry comprising *function* of arguments stream and object to print; and *priority* as *type* into *table*. If *function* is NIL, remove *type* from *table*. Return NIL.

(*f***pprint-dispatch** *foo* [*table*[*v****print-pprint-dispatch***]])
▷ Return highest priority *function* associated with type of *foo* and T if there was a matching type specifier in *table*.

(*f***copy-pprint-dispatch** [*table*[*v****print-pprint-dispatch***]])
▷ Return copy of *table* or, if *table* is NIL, initial value of *v****print-pprint-dispatch***.

*v****print-pprint-dispatch*** ▷ Current pretty print dispatch table.

13.5 Format

(*m***formatter** $\widetilde{control}$)
▷ Return function of *stream* and *arg** applying *f***format** to *stream*, *control*, and *arg** returning NIL or any excess *arg*s.

(*f***format** {T|NIL|*out-string*|*out-stream*} *control arg**)
▷ Output string *control* which may contain ~ directives possibly taking some *arg*s. Alternatively, *control* can be a function returned by *m***formatter** which is then applied to *out-stream* and *arg**. Output to *out-string*, *out-stream* or, if first argument is T, to *v****standard-output***. Return NIL. If first argument is NIL, return formatted output.

~ [*min-col*[0]] [,[*col-inc*[1]] [,[*min-pad*[0]] [,'*pad-char*[ ]]]] [:] [@] {**A**|**S**}
▷ **Aesthetic/Standard.** Print argument of any type for consumption by humans/by the reader, respectively. With **:**, print NIL as () rather than nil; with **@**, add *pad-char*s on the left rather than on the right.

~ [*radix*[10]] [,[*width*] [,['*pad-char*[ ]] [,['*comma-char*[,]] [,*comma-interval*[3]]]]] [:] [@] **R**
▷ **Radix.** (With one or more prefix arguments.) Print argument as number; with **:**, group digits *comma-interval* each; with **@**, always prepend a sign.

{~**R**|~**:R**|~**@R**|~**@:R**}
▷ **Roman.** Take argument as number and print it as English cardinal number, as English ordinal number, as Roman numeral, or as old Roman numeral, respectively.

*v****read-suppress***[NIL]
▷ If T, reader is syntactically more tolerant.

(*f***set-macro-character** *char function* [*non-term-p*[NIL] [$\widetilde{rt}$[*v****readtable***]]])
▷ Make *char* a macro character associated with *function* of stream and *char*. Return T.

(*f***get-macro-character** *char* [*rt*[*v****readtable***]])
▷ Reader macro function associated with *char*, and T if *char* is a non-terminating macro character.

(*f***make-dispatch-macro-character** *char* [*non-term-p*[NIL] [*rt*[*v****readtable***]]])
▷ Make *char* a dispatching macro character. Return T.

(*f***set-dispatch-macro-character** *char sub-char function* [$\widetilde{rt}$[*v****readtable***]])
▷ Make *function* of stream, *n*, *sub-char* a dispatch function of *char* followed by *n*, followed by *sub-char*. Return T.

(*f***get-dispatch-macro-character** *char sub-char* [*rt*[*v****readtable***]])
▷ Dispatch function associated with *char* followed by *sub-char*.

13.3 Character Syntax

#| *multi-line-comment** **|#**
; *one-line-comment**
▷ Comments. There are stylistic conventions:

;;;; *title*	▷ Short title for a block of code.
;;; *intro*	▷ Description before a block of code.
;; *state*	▷ State of program or of following code.
;*explanation* **;** *continuation*	▷ Regarding line on which it appears.

(*foo**[**.** *bar*[NIL]**)** ▷ List of *foo*s with the terminating cdr *bar*.

" ▷ Begin and end of a string.

'*foo* ▷ (*s***quote** *foo*); *foo* unevaluated.

`([*foo*] [**,***bar*] [**,@***baz*] [**,.**$\widetilde{quux}$] [*bing*])
▷ Backquote. *s***quote** *foo* and *bing*; evaluate *bar* and splice the lists *baz* and *quux* into their elements. When nested, outermost commas inside the innermost backquote expression belong to this backquote.

#*c* ▷ (*f***character "***c***"**), the character *c*.

#B*n*; **#O***n*; *n.*; **#X***n*; **#***r***R***n*
▷ Integer of radix 2, 8, 10, 16, or *r*; $2 \le r \le 36$.

*n***/***d* ▷ The **ratio** $\frac{n}{d}$.

$\Big\{[m].n[\{\mathbf{S}|\mathbf{F}|\mathbf{D}|\mathbf{L}|\mathbf{E}\}x_{[E0]}]\,\Big|\,m[.[n]]\{\mathbf{S}|\mathbf{F}|\mathbf{D}|\mathbf{L}|\mathbf{E}\}x\Big\}$
▷ $m.n \cdot 10^x$ as **short-float**, **single-float**, **double-float**, **long-float**, or the type from ***read-default-float-format***.

#C(*a b***)** ▷ (*f***complex** *a b*), the complex number $a + b$i.

#'*foo* ▷ (*s***function** *foo*); the function named *foo*.

#*n***A***sequence* ▷ *n*-dimensional array.

#[*n*]**(***foo**)
▷ Vector of some (or *n*) *foo*s filled with last *foo* if necessary.

#[*n*]*****b*
▷ Bit vector of some (or *n*) *b*s filled with last *b* if necessary.

#S(*type* {*slot value*}***)** ▷ Structure of *type*.

#P*string* ▷ A pathname.

#:*foo* ▷ Uninterned symbol *foo*.

#.*form* ▷ Read-time value of *form*.

$_{v}$***read-eval*** T ▷ If NIL, a **reader-error** is signalled at **#.**.

#$integer$**=** foo ▷ Give foo the label $integer$.

#$integer$**#** ▷ Object labelled $integer$.

#< ▷ Have the reader signal **reader-error**.

#+*feature when-feature*
#−*feature unless-feature*
▷ Means *when-feature* if *feature* is T; means *unless-feature* if *feature* is NIL. *feature* is a symbol from $_{v}$***features***, or ({**and** | **or**} *feature**), or (**not** *feature*).

$_{v}$***features***
▷ List of symbols denoting implementation-dependent features.

|c^*|; \c
▷ Treat arbitrary character(s) c as alphabetic preserving case.

13.4 Printer

($_{f}$**prin1** | $_{f}$**print** | $_{f}$**pprint** | $_{f}$**princ** foo [$\widetilde{stream}$ $_{v}$***standard-output***])
▷ Print foo to $stream$ $_{f}$**read**ably, $_{f}$**read**ably between a newline and a space, $_{f}$**read**ably after a newline, or human-readably without any extra characters, respectively. $_{f}$**prin1**, $_{f}$**print** and $_{f}$**princ** return foo.

($_{f}$**prin1-to-string** foo)
($_{f}$**princ-to-string** foo)
▷ Print foo to $string$ $_{f}$**read**ably or human-readably, respectively.

($_{g}$**print-object** $object$ $\widetilde{stream}$)
▷ Print $object$ to $stream$. Called by the Lisp printer.

($_{m}$**print-unreadable-object** (foo $\widetilde{stream}$ {**:type** $bool$ NIL | **:identity** $bool$ NIL}) $form^{P*}$)
▷ Enclosed in **#<** and **>**, print foo by means of $form$s to $stream$. Return NIL.

($_{f}$**terpri** [$\widetilde{stream}$ $_{v}$***standard-output***])
▷ Output a newline to $stream$. Return NIL.

($_{f}$**fresh-line** [$\widetilde{stream}$ $_{v}$***standard-output***])
▷ Output a newline to $stream$ and return T unless $stream$ is already at the start of a line.

($_{f}$**write-char** $char$ [$\widetilde{stream}$ $_{v}$***standard-output***])
▷ Output $char$ to $stream$.

($_{f}$**write-string** | $_{f}$**write-line** $string$ [$\widetilde{stream}$ $_{v}$***standard-output*** [{**:start** $start$ 0 | **:end** end NIL}]])
▷ Write $string$ to $stream$ without/with a trailing newline.

($_{f}$**write-byte** $byte$ $\widetilde{stream}$) ▷ Write $byte$ to binary $stream$.

($_{f}$**write-sequence** $sequence$ $\widetilde{stream}$ {**:start** $start$ 0 | **:end** end NIL})
▷ Write elements of $sequence$ to binary or character $stream$.

($_{f}$**write** | $_{f}$**write-to-string** foo {**:array** $bool$ | **:base** $radix$ | **:case** {**:upcase** | **:downcase** | **:capitalize**} | **:circle** $bool$ | **:escape** $bool$ | **:gensym** $bool$ | **:length** {int | NIL} | **:level** {int | NIL} | **:lines** {int | NIL} | **:miser-width** {int | NIL} | **:pprint-dispatch** $dispatch\text{-}table$ | **:pretty** $bool$ | **:radix** $bool$ | **:readably** $bool$ | **:right-margin** {int | NIL} | **:stream** $\widetilde{stream}$ $_{v}$***standard-output***})
▷ Print foo to $stream$ and return foo, or print foo into string, respectively, after dynamically setting printer variables corresponding to keyword parameters (***print-**bar***** becoming **:**bar). (**:stream** keyword with $_{f}$**write** only.)

($_{f}$**pprint-fill** $\widetilde{stream}$ foo [$parenthesis$ T [$noop$]])
($_{f}$**pprint-tabular** $\widetilde{stream}$ foo [$parenthesis$ T [$noop$ [n 16]]])
($_{f}$**pprint-linear** $\widetilde{stream}$ foo [$parenthesis$ T [$noop$]])
▷ Print foo to $stream$. If foo is a list, print as many elements per line as possible; do the same in a table with a column width of n ems; or print either all elements on one line or each on its own line, respectively. Return NIL. Usable with $_{f}$**format** directive **~//**.

($_{m}$**pprint-logical-block** ($\widetilde{stream}$ $list$ {{**:prefix** $string$ | **:per-line-prefix** $string$} | **:suffix** $string$ "")
(**declare** $\widetilde{decl^*})^*$ $form^{P*}$)
▷ Evaluate $form$s, which should print $list$, with $stream$ locally bound to a pretty printing stream which outputs to the original $stream$. If $list$ is in fact not a list, it is printed by $_{f}$**write**. Return NIL.

($_{m}$**pprint-pop**)
▷ Take next element off $list$. If there is no remaining tail of $list$, or $_{v}$***print-length*** or $_{v}$***print-circle*** indicate printing should end, send element together with an appropriate indicator to $stream$.

($_{f}$**pprint-tab** {**:line** | **:line-relative** | **:section** | **:section-relative**} c i [$\widetilde{stream}$ $_{v}$***standard-output***])
▷ Move cursor forward to column number $c + ki$, $k \geq 0$ being as small as possible.

($_{f}$**pprint-indent** {**:block** | **:current**} n [$\widetilde{stream}$ $_{v}$***standard-output***])
▷ Specify indentation for innermost logical block relative to leftmost position/to current position. Return NIL.

($_{m}$**pprint-exit-if-list-exhausted**)
▷ If $list$ is empty, terminate logical block. Return NIL otherwise.

($_{f}$**pprint-newline** {**:linear** | **:fill** | **:miser** | **:mandatory**} [$\widetilde{stream}$ $_{v}$***standard-output***])
▷ Print a conditional newline if $stream$ is a pretty printing stream. Return NIL.

$_{v}$***print-array*** ▷ If T, print arrays $_{f}$**read**ably.

$_{v}$***print-base*** 10 ▷ Radix for printing rationals, from 2 to 36.

$_{v}$***print-case*** :upcase
▷ Print symbol names all uppercase (**:upcase**), all lowercase (**:downcase**), capitalized (**:capitalize**).

(*f*stream-external-format *stream*)
▷ External file format designator.

*v*terminal-io* ▷ Bidirectional stream to user terminal.

*v*standard-input*
*v*standard-output*
*v*error-output*
▷ Standard input stream, standard output stream, or standard error output stream, respectively.

*v*debug-io*
*v*query-io*
▷ Bidirectional streams for debugging and user interaction.

13.7 Pathnames and Files

(*f*make-pathname
{ :host {*host*|NIL|:unspecific}
:device {*device*|NIL|:unspecific}
:directory {{*directory*|:wild|NIL|:unspecific}
({:absolute|:relative} {*directory*|:wild|:wild-inferiors|:up|:back}*)}
:name {*file-name*|:wild|NIL|:unspecific}
:type {*file-type*|:wild|NIL|:unspecific}
:version {:newest|*version*|:wild|NIL|:unspecific}
:defaults *path*[host from *v*default-pathname-defaults*]
:case {:local|:common}[:local] }
▷ Construct a logical pathname if there is a logical pathname translation for *host*, otherwise construct a physical pathname. For **:case :local**, leave case of components unchanged. For **:case :common**, leave mixed-case components unchanged; convert all-uppercase components into local customary case; do the opposite with all-lowercase components.

({*f*pathname-host|*f*pathname-device|*f*pathname-directory|*f*pathname-name|*f*pathname-type} *path-or-stream* [:case {:local|:common}[:local]])
(*f*pathname-version *path-or-stream*)
▷ Return pathname component.

(*f*parse-namestring *foo* [*host*
[*default-pathname*[*v*default-pathname-defaults*]
{ :start *start*[0]
:end *end*[NIL]
:junk-allowed *bool*[NIL] }]])
▷ Return pathname converted from string, pathname, or stream *foo*; and position$_2$ where parsing stopped.

(*f*merge-pathnames *path-or-stream*
[*default-path-or-stream*[*v*default-pathname-defaults*]
[*default-version*[:newest]]])
▷ Return pathname made by filling in components missing in *path-or-stream* from *default-path-or-stream*.

*v*default-pathname-defaults*
▷ Pathname to use if one is needed and none supplied.

(*f*user-homedir-pathname [*host*]) ▷ User's home directory.

(*f*enough-namestring *path-or-stream*
[*root-path*[*v*default-pathname-defaults*]])
▷ Return minimal path string that sufficiently describes the path of *path-or-stream* relative to *root-path*.

(*f*namestring *path-or-stream*)
(*f*file-namestring *path-or-stream*)
(*f*directory-namestring *path-or-stream*)
(*f*host-namestring *path-or-stream*)
▷ Return string representing full pathname; name, type, and version; directory name; or host name, respectively, of *path-or-stream*.

~ [*width*] [,['*pad-char*[␣]] [,['*comma-char*[,]] [,*comma-interval*[3]]]] [:] [@] {D|B|O|X}
▷ **Decimal/Binary/Octal/Hexadecimal.** Print integer argument as number. With **:**, group digits *comma-interval* each; with **@**, always prepend a sign.

~ [*width*] [,[*dec-digits*] [,[*shift*[0]] [,['*overflow-char*] [,'*pad-char*[␣]]]] [@] F
▷ **Fixed-Format Floating-Point.** With **@**, always prepend a sign.

~ [*width*] [,[*dec-digits*] [,[*exp-digits*] [,[*scale-factor*[1]] [,['*overflow-char*] [,['*pad-char*[␣]] [,'*exp-char*]]]]]] [@] {E|G}
▷ **Exponential/General Floating-Point.** Print argument as floating-point number with *dec-digits* after decimal point and *exp-digits* in the signed exponent. With **~G**, choose either **~E** or **~F**. With **@**, always prepend a sign.

~ [*dec-digits*[2]] [,[*int-digits*[1]] [,[*width*[0]] [,'*pad-char*[␣]]]] [:] [@] $
▷ **Monetary Floating-Point.** Print argument as fixed-format floating-point number. With **:**, put sign before any padding; with **@**, always prepend a sign.

{~C|~:C|~@C|~@:C}
▷ **Character.** Print, spell out, print in #\ syntax, or tell how to type, respectively, argument as (possibly non-printing) character.

{~(*text* ~)|~:(*text* ~)|~@(*text* ~)|~@:(*text* ~)}
▷ **Case-Conversion.** Convert *text* to lowercase, convert first letter of each word to uppercase, capitalize first word and convert the rest to lowercase, or convert to uppercase, respectively.

{~P|~:P|~@P|~@:P}
▷ **Plural.** If argument **eql** 1 print nothing, otherwise print s; do the same for the previous argument; if argument **eql** 1 print y, otherwise print ies; do the same for the previous argument, respectively.

~ [$n_{[1]}$] % ▷ **Newline.** Print n newlines.

~ [$n_{[1]}$] &
▷ **Fresh-Line.** Print $n-1$ newlines if output stream is at the beginning of a line, or n newlines otherwise.

{~␣|~:␣|~@␣|~@:␣}
▷ **Conditional Newline.** Print a newline like **pprint-newline** with argument **:linear**, **:fill**, **:miser**, or **:mandatory**, respectively.

{~:↩|~@↩|~↩}
▷ **Ignored Newline.** Ignore newline, or whitespace following newline, or both, respectively.

~ [$n_{[1]}$] | ▷ **Page.** Print n page separators.

~ [$n_{[1]}$] ~ ▷ **Tilde.** Print n tildes.

~ [*min-col*[0]] [,[*col-inc*[1]] [,[*min-pad*[0]] [,'*pad-char*[␣]]]] [:] [@] < [*nl-text* ~[*spare*[0] [,*width*]]:;] {*text* ~;}* *text* ~>
▷ **Justification.** Justify text produced by *text*s in a field of at least *min-col* columns. With **:**, right justify; with **@**, left justify. If this would leave less than *spare* characters on the current line, output *nl-text* first.

~ [:] [@] < {[*prefix*[""] ~;]|[*per-line-prefix* ~@;]} *body* [~; *suffix*[""] ~:] [@] >
▷ **Logical Block.** Act like **pprint-logical-block** using *body* as *f*format control string on the elements of the list argument or, with **@**, on the remaining arguments, which are extracted by **pprint-pop**. With **:**, *prefix* and *suffix* default to (and). When closed by **~@:>**, spaces in *body* are replaced with conditional newlines.

{~ [$n_{[0]}$] i|~ [$n_{[0]}$] :i}
▷ **Indent.** Set indentation to n relative to leftmost/to current position.

~ [c_{1}] [,i_{1}] [:] [@] **T**
▷ **Tabulate.** Move cursor forward to column number $c + ki$, $k \geq 0$ being as small as possible. With **:**, calculate column numbers relative to the immediately enclosing section. With **@**, move to column number $c_0 + c + ki$ where c_0 is the current position.

{~ [m_{1}] ***** | ~ [m_{1}] **:*** | ~ [n_{0}] **@***}
▷ **Go-To.** Jump m arguments forward, or backward, or to argument n.

~ [*limit*] [:] [@] **{** *text* **~}**
▷ **Iteration.** Use *text* repeatedly, up to *limit*, as control string for the elements of the list argument or (with **@**) for the remaining arguments. With **:** or **@:**, list elements or remaining arguments should be lists of which a new one is used at each iteration step.

~ [*x* [,*y* [,*z*]]] ^
▷ **Escape Upward.** Leave immediately **~<** **~>**, **~<** **~:>**, **~{** **~}**, **~?**, or the entire *f*-**format** operation. With one to three prefixes, act only if $x = 0$, $x = y$, or $x \leq y \leq z$, respectively.

~ [*i*] [:] [@] **[** [{*text* **~;**}* *text*] [**~:;** *default*] **~]**
▷ **Conditional Expression.** Use the zero-indexed argumenth (or *i*th if given) *text* as a *f*-**format** control subclause. With **:**, use the first *text* if the argument value is `NIL`, or the second *text* if it is `T`. With **@**, do nothing for an argument value of `NIL`. Use the only *text* and leave the argument to be read again if it is `T`.

{**~?** | **~@?**}
▷ **Recursive Processing.** Process two arguments as control string and argument list, or take one argument as control string and use then the rest of the original arguments.

~ [*prefix* {,*prefix*}*] [:] [@] **/**[*package* [:]:`cl-user:`]*function***/**
▷ **Call Function.** Call all-uppercase *package*::*function* with the arguments stream, format-argument, colon-p, at-sign-p and *prefix*es for printing format-argument.

~ [:] [@] **W**
▷ **Write.** Print argument of any type obeying every printer control variable. With **:**, pretty-print. With **@**, print without limits on length or depth.

{**V** | **#**}
▷ In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.

13.6 Streams

(*f*-**open** *path*
- **:direction** { **:input** | **:output** | **:io** | **:probe** } [**:input**]
- **:element-type** { *type* | **:default** } [`character`]
- **:if-exists** { **:new-version** | **:error** | **:rename** | **:rename-and-delete** | **:overwrite** | **:append** | **:supersede** | `NIL` } [**:new-version** if *path* specifies **:newest**; `NIL` otherwise]
- **:if-does-not-exist** { **:error** | **:create** | `NIL` } [`NIL` for **:direction** **:probe**; {**:create** | **:error**} otherwise]
- **:external-format** *format* [**:default**]

)

▷ Open **file-stream** to *path*.

(*f*-**make-concatenated-stream** *input-stream**)
(*f*-**make-broadcast-stream** *output-stream**)
(*f*-**make-two-way-stream** *input-stream-part output-stream-part*)
(*f*-**make-echo-stream** *from-input-stream to-output-stream*)
(*f*-**make-synonym-stream** *variable-bound-to-stream*)

▷ Return stream of indicated type.

(*f*-**make-string-input-stream** *string* [*start*$_{0}$ [*end*$_{\texttt{NIL}}$]])
▷ Return a **string-stream** supplying the characters from *string*.

(*f*-**make-string-output-stream** [**:element-type** *type*$_{\texttt{character}}$])
▷ Return a **string-stream** accepting characters (available via *f*-**get-output-stream-string**).

(*f*-**concatenated-stream-streams** *concatenated-stream*)
(*f*-**broadcast-stream-streams** *broadcast-stream*)
▷ Return list of streams *concatenated-stream* still has to read from/*broadcast-stream* is broadcasting to.

(*f*-**two-way-stream-input-stream** *two-way-stream*)
(*f*-**two-way-stream-output-stream** *two-way-stream*)
(*f*-**echo-stream-input-stream** *echo-stream*)
(*f*-**echo-stream-output-stream** *echo-stream*)
▷ Return source stream or sink stream of *two-way-stream*/*echo-stream*, respectively.

(*f*-**synonym-stream-symbol** *synonym-stream*)
▷ Return symbol of *synonym-stream*.

(*f*-**get-output-stream-string** $\widetilde{\textit{string-stream}}$)
▷ Clear and return as a string characters on *string-stream*.

(*f*-**file-position** *stream* [{ **:start** | **:end** | *position* }])
▷ Return position within stream, or set it to *position* and return `T` on success.

(*f*-**file-string-length** *stream foo*)
▷ Length *foo* would have in *stream*.

(*f*-**listen** [*stream*$_{v\text{*standard-input*}}$])
▷ `T` if there is a character in input *stream*.

(*f*-**clear-input** [$\widetilde{\textit{stream}}$$_{v\text{*standard-input*}}$])
▷ Clear input from *stream*, return `NIL`.

({ *f*-**clear-output** | *f*-**force-output** | *f*-**finish-output** } [$\widetilde{\textit{stream}}$$_{v\text{*standard-output*}}$])
▷ End output to *stream* and return `NIL` immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.

(*f*-**close** $\widetilde{\textit{stream}}$ [**:abort** *bool*$_{\texttt{NIL}}$])
▷ Close *stream*. Return `T` if *stream* had been open. If **:abort** is `T`, delete associated file.

(*m*-**with-open-file** (*stream path open-arg**) (**declare** $\widehat{\textit{decl}}$*)* *form*^{P}*)
▷ Use *f*-**open** with *open-arg*s to temporarily create *stream* to *path*; return values of *form*s.

(*m*-**with-open-stream** (*foo* $\widetilde{\textit{stream}}$) (**declare** $\widehat{\textit{decl}}$*)* *form*^{P}*)
▷ Evaluate *form*s with *foo* locally bound to *stream*. Return values of *form*s.

(*m*-**with-input-from-string** (*foo string* { **:index** $\widetilde{\textit{index}}$ | **:start** *start*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ }) (**declare** $\widehat{\textit{decl}}$*)* *form*^{P}*)
▷ Evaluate *form*s with *foo* locally bound to input **string-stream** from *string*. Return values of *form*s; store next reading position into *index*.

(*m*-**with-output-to-string** (*foo* [$\widetilde{\textit{string}}$$_{\texttt{NIL}}$ [**:element-type** *type*$_{\texttt{character}}$]]) (**declare** $\widehat{\textit{decl}}$*)* *form*^{P}*)
▷ Evaluate *form*s with *foo* locally bound to an output **string-stream**. Append output to *string* and return values of *form*s if *string* is given. Return string containing output otherwise.

14.4 Standard Packages

common-lisp|**cl**
▷ Exports the defined names of Common Lisp except for those in the **keyword** package.

common-lisp-user|**cl-user**
▷ Current package after startup; uses package **common-lisp**.

keyword
▷ Contains symbols which are defined to be of type **keyword**.

15 Compiler

15.1 Predicates

(*f* **special-operator-p** *foo*) ▷ T if *foo* is a special operator.

(*f* **compiled-function-p** *foo*)
▷ T if *foo* is of type **compiled-function**.

15.2 Compilation

(*f* **compile** { NIL *definition* | { *name* | (**setf** *name*) } [*definition*] })
▷ Return compiled function or replace *name*'s function definition with the compiled function. Return T in case of **warning**s or **error**s, and T (2) in case of **warning**s or **error**s excluding **style-warning**s.

(*f* **compile-file** *file* { | **:output-file** *out-path* | **:verbose** *bool* [*v* ***compile-verbose***] | **:print** *bool* [*v* ***compile-print***] | **:external-format** *file-format* [**:default**] })
▷ Write compiled contents of *file* to *out-path*. Return true output path or NIL, T (2) in case of **warning**s or **error**s, T (3) in case of **warning**s or **error**s excluding **style-warning**s.

(*f* **compile-file-pathname** *file* [**:output-file** *path*] [*other-keyargs*])
▷ Pathname *f* **compile-file** writes to if invoked with the same arguments.

(*f* **load** *path* { | **:verbose** *bool* [*v* ***load-verbose***] | **:print** *bool* [*v* ***load-print***] | **:if-does-not-exist** *bool* [T] | **:external-format** *file-format* [**:default**] })
▷ Load source file or compiled file into Lisp environment. Return T if successful.

v ***compile-file*** / *v* ***load*** - { **pathname*** [NIL] | **truename*** [NIL] }
▷ Input file used by *f* **compile-file**/by *f* **load**.

v ***compile*** / *v* ***load*** - { **print*** | **verbose*** }
▷ Defaults used by *f* **compile-file**/by *f* **load**.

(*s* **eval-when** ({ | {**:compile-toplevel**|**compile**} | {**:load-toplevel**|**load**} | {**:execute**|**eval**} }) *form*P*)
▷ Return values of *forms* if *s* **eval-when** is in the top-level of a file being compiled, in the top-level of a compiled file being loaded, or anywhere, respectively. Return NIL if *form*s are not evaluated. (**compile**, **load** and **eval** deprecated.)

(*s* **locally** (**declare** $\widehat{\textit{decl}^*}$)* *form*P*)
▷ Evaluate *form*s in a lexical environment with declarations *decl* in effect. Return values of *form*s.

(*m* **with-compilation-unit** ([**:override** *bool* [NIL]]) *form*P*)
▷ Return values of *form*s. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of *form*s.

(*f* **translate-pathname** *path-or-stream wildcard-path-a wildcard-path-b*)
▷ Translate the path of *path-or-stream* from *wildcard-path-a* into *wildcard-path-b*. Return new path.

(*f* **pathname** *path-or-stream*) ▷ Pathname of *path-or-stream*.

(*f* **logical-pathname** *logical-path-or-stream*)
▷ Logical pathname of *logical-path-or-stream*. Logical pathnames are represented as all-uppercase "[*host*:][;]{ { *dir* | * }+ | ** };*{ *name* | * }* [. { { *type* | * }+ | LISP } [. { *version* | * | newest | NEWEST }]]".

(*f* **logical-pathname-translations** *logical-host*)
▷ List of (*from-wildcard to-wildcard*) translations for *logical-host*. **setf**able.

(*f* **load-logical-pathname-translations** *logical-host*)
▷ Load *logical-host*'s translations. Return NIL if already loaded; return T if successful.

(*f* **translate-logical-pathname** *path-or-stream*)
▷ Physical pathname corresponding to (possibly logical) pathname of *path-or-stream*.

(*f* **probe-file** *file*)
(*f* **truename** *file*)
▷ Canonical name of *file*. If *file* does not exist, return NIL/signal **file-error**, respectively.

(*f* **file-write-date** *file*) ▷ Time at which *file* was last written.

(*f* **file-author** *file*) ▷ Return name of *file* owner.

(*f* **file-length** *stream*) ▷ Return length of *stream*.

(*f* **rename-file** *foo bar*)
▷ Rename file *foo* to *bar*. Unspecified components of path *bar* default to those of *foo*. Return new pathname, old physical file name (2), and new physical file name (3).

(*f* **delete-file** *file*) ▷ Delete *file*. Return T.

(*f* **directory** *path*) ▷ List of pathnames matching *path*.

(*f* **ensure-directories-exist** *path* [**:verbose** *bool*])
▷ Create parts of *path* if necessary. Second return value is T (2) if something has been created.

14 Packages and Symbols

The Loop Facility provides additional means of symbol handling; see **loop**, page 21.

14.1 Predicates

(*f* **symbolp** *foo*)
(*f* **packagep** *foo*) ▷ T if *foo* is of indicated type.
(*f* **keywordp** *foo*)

14.2 Packages

:*bar*|**keyword:***bar* ▷ Keyword, evaluates to :*bar*.

package:*symbol* ▷ Exported *symbol* of *package*.

package::*symbol* ▷ Possibly unexported *symbol* of *package*.

(*m*defpackage *foo* { (:nicknames *nick**)* | (:documentation *string*) | (:intern *interned-symbol**)* | (:use *used-package**)* | (:import-from *pkg imported-symbol**)* | (:shadowing-import-from *pkg shd-symbol**)* | (:shadow *shd-symbol**)* | (:export *exported-symbol**)* | (:size *int*) })

▷ Create or modify package *foo* with *interned-symbol*s, symbols from *used-package*s, *imported-symbol*s, and *shd-symbol*s. Add *shd-symbol*s to *foo*'s shadowing list.

(*f*make-package *foo* { :nicknames (*nick**) NIL | :use (*used-package**) })

▷ Create package *foo*.

(*f*rename-package *package new-name* [*new-nicknames* NIL])

▷ Rename *package*. Return renamed package.

(*m*in-package $\widehat{foo}$) ▷ Make package *foo* current.

({*f*use-package | *f*unuse-package} *other-packages* [*package* *v**package*])

▷ Make exported symbols of *other-packages* available in *package*, or remove them from *package*, respectively. Return T.

(*f*package-use-list *package*)
(*f*package-used-by-list *package*)

▷ List of other packages used by/using *package*.

(*f*delete-package $\widetilde{package}$)

▷ Delete *package*. Return T if successful.

*v**package* common-lisp-user ▷ The current package.

(*f*list-all-packages) ▷ List of registered packages.

(*f*package-name *package*) ▷ Name of *package*.

(*f*package-nicknames *package*) ▷ Nicknames of *package*.

(*f*find-package *name*) ▷ Package with *name* (case-sensitive).

(*f*find-all-symbols *foo*)

▷ List of symbols *foo* from all registered packages.

({*f*intern | *f*find-symbol} *foo* [*package* *v**package*])

▷ Intern or find, respectively, symbol *foo* in *package*. Second return value is one of :internal, :external, or :inherited (or NIL if *f*intern has created a fresh symbol).

(*f*unintern *symbol* [*package* *v**package*])

▷ Remove *symbol* from *package*, return T on success.

({*f*import | *f*shadowing-import} *symbols* [*package* *v**package*])

▷ Make *symbols* internal to *package*. Return T. In case of a name conflict signal correctable **package-error** or shadow the old symbol, respectively.

(*f*shadow *symbols* [*package* *v**package*])

▷ Make *symbols* of *package* shadow any otherwise accessible, equally named symbols from other packages. Return T.

(*f*package-shadowing-symbols *package*)

▷ List of symbols of *package* that shadow any otherwise accessible, equally named symbols from other packages.

(*f*export *symbols* [*package* *v**package*])

▷ Make *symbols* external to *package*. Return T.

(*f*unexport *symbols* [*package* *v**package*])

▷ Revert *symbols* to internal status. Return T.

({*m*do-symbols | *m*do-external-symbols | *m*do-all-symbols} ($\widehat{var}$ [*package* *v**package* [*result* NIL]]) (declare $\widehat{decl}$*)* { $\widehat{tag}$ | *form* }*)

▷ Evaluate *s*tagbody-like body with *var* successively bound to every symbol from *package*, to every external symbol from *package*, or to every symbol from all registered packages, respectively. Return values of *result*. Implicitly, the whole form is a *s*block named NIL.

(*m*with-package-iterator (*foo packages* [:internal | :external | :inherited]) (declare $\widehat{decl}$*)* *form*P*)

▷ Return values of *form*s. In *form*s, successive invocations of (*foo*) return: T if a symbol is returned; a symbol from *packages*; accessibility (:internal, :external, or :inherited); and the package the symbol belongs to.

(*f*require *module* [*paths* NIL])

▷ If not in *v**modules*, try *paths* to load *module* from. Signal **error** if unsuccessful. Deprecated.

(*f*provide *module*)

▷ If not already there, add *module* to *v**modules*. Deprecated.

*v**modules* ▷ List of names of loaded modules.

14.3 Symbols

A **symbol** has the attributes *name*, home **package**, property list, and optionally value (of global constant or variable *name*) and function (**function**, macro, or special operator *name*).

(*f*make-symbol *name*)

▷ Make fresh, uninterned symbol *name*.

(*f*gensym [*s* G])

▷ Return fresh, uninterned symbol #:*sn* with *n* from *v**gensym-counter*. Increment *v**gensym-counter*.

(*f*gentemp [*prefix* T [*package* *v**package*]])

▷ Intern fresh symbol in package. Deprecated.

(*f*copy-symbol *symbol* [*props* NIL])

▷ Return uninterned copy of *symbol*. If *props* is T, give copy the same value, function and property list.

(*f*symbol-name *symbol*)
(*f*symbol-package *symbol*)
(*f*symbol-plist *symbol*)
(*f*symbol-value *symbol*)
(*f*symbol-function *symbol*)

▷ Name, package, property list, value, or function, respectively, of *symbol*. **setf**able.

({*g*documentation | (**setf** *g*documentation)} *new-doc*) *foo* { 'variable | 'function | 'compiler-macro | 'method-combination | 'structure | 'type | 'setf | T })

▷ Get/set documentation string of *foo* of given type.

*c*t

▷ Truth; the supertype of every type including **t**; the superclass of every class except **t**; *v**terminal-io*.

*c*nil | *c*()

▷ Falsity; the empty list; the empty type, subtype of every type; *v**standard-input*; *v**standard-output*; the global environment.

Index

(*s*load-time-value *form* [*read-only*NIL])
▷ Evaluate *form* at compile time and treat its value as literal at run time.

(*s*quote *foo*) ▷ Return unevaluated *foo*.

(*g*make-load-form *foo* [*environment*])
▷ Its methods are to return a creation form which on evaluation at *f*load time returns an object equivalent to *foo*, and an optional initialization form which on evaluation performs some initialization of the object.

(*f*make-load-form-saving-slots *foo* {:slot-names *slots*all local slots | :environment *environment*})
▷ Return a creation form and an initialization form which on evaluation construct an object equivalent to *foo* with *slots* initialized with the corresponding values from *foo*.

(*f*macro-function *symbol* [*environment*])
(*f*compiler-macro-function {*name* | (setf *name*)} [*environment*])
▷ Return specified macro function, or compiler macro function, respectively, if any. Return NIL otherwise. **setf**able.

(*f*eval *arg*)
▷ Return values of value of *arg* evaluated in global environment.

15.3 REPL and Debugging

v+ | *v*++ | *v*+++
*v** | *v*** | *v****
v/ | *v*// | *v*///
▷ Last, penultimate, or antepenultimate form evaluated in the REPL, or their respective primary value, or a list of their respective values.

v- ▷ Form currently being evaluated by the REPL.

(*f*apropos *string* [*package*NIL])
▷ Print interned symbols containing *string*.

(*f*apropos-list *string* [*package*NIL])
▷ List of interned symbols containing *string*.

(*f*dribble [*path*])
▷ Save a record of interactive session to file at *path*. Without *path*, close that file.

(*f*ed [*file-or-function*NIL]) ▷ Invoke editor if possible.

({*f*macroexpand-1 | *f*macroexpand} *form* [*environment*NIL])
▷ Return macro expansion, once or entirely, respectively, of *form* and T if *form* was a macro form. Return *form* and NIL otherwise.

*v**macroexpand-hook*
▷ Function of arguments expansion function, macro form, and environment called by *f*macroexpand-1 to generate macro expansions.

(*m*trace {*function* | (setf *function*)}*)
▷ Cause *function*s to be traced. With no arguments, return list of traced functions.

(*m*untrace {*function* | (setf *function*)}*)
▷ Stop *function*s, or each currently traced function, from being traced.

*v**trace-output*
▷ Output stream *m*trace and *m*time send their output to.

(*m*step *form*)
▷ Step through evaluation of *form*. Return values of *form*.

(*f* **break** [*control arg**])
▷ Jump directly into debugger; return NIL. See page 36, *f* **format**, for *control* and *args*.

(*m* **time** *form*)
▷ Evaluate *form*s and print timing information to *v* ∗**trace-output**∗. Return values of *form*.

(*f* **inspect** *foo*) ▷ Interactively give information about *foo*.

(*f* **describe** *foo* [*stream*$_{v*standard-output*}$])
▷ Send information about *foo* to *stream*.

(*g* **describe-object** *foo* [*stream*])
▷ Send information about *foo* to *stream*. Called by *f* **describe**.

(*f* **disassemble** *function*)
▷ Send disassembled representation of *function* to *v* ∗**standard-output**∗. Return NIL.

(*f* **room** [{NIL | **:default** | T}$_{:default}$])
▷ Print information about internal storage management to ∗**standard-output**∗.

15.4 Declarations

(*f* **proclaim** *decl*)
(*m* **declaim** *decl**)
▷ Globally make declaration(s) *decl*. *decl* can be: **declaration**, **type**, **ftype**, **inline**, **notinline**, **optimize**, or **special**. See below.

(**declare** *decl**)
▷ Inside certain forms, locally make declarations *decl**. *decl* can be: **dynamic-extent**, **type**, **ftype**, **ignorable**, **ignore**, **inline**, **notinline**, **optimize**, or **special**. See below.

(**declaration** foo*)
▷ Make *foo*s names of declarations.

(**dynamic-extent** *variable** (**function** *function*)*)
▷ Declare lifetime of *variable*s and/or *function*s to end when control leaves enclosing block.

([**type**] *type variable**)
(**ftype** *type function**)
▷ Declare *variable*s or *function*s to be of *type*.

({**ignorable** | **ignore**} {*var* | (**function** *function*)}*)
▷ Suppress warnings about used/unused bindings.

(**inline** *function**)
(**notinline** *function**)
▷ Tell compiler to integrate/not to integrate, respectively, called *function*s into the calling routine.

(**optimize** { **compilation-speed** | (**compilation-speed** $n_{[3]}$) | **debug** | (**debug** $n_{[3]}$) | **safety** | (**safety** $n_{[3]}$) | **space** | (**space** $n_{[3]}$) | **speed** | (**speed** $n_{[3]}$) })
▷ Tell compiler how to optimize. $n = 0$ means unimportant, $n = 1$ is neutral, $n = 3$ means important.

(**special** *var**) ▷ Declare *var*s to be dynamic.

16 External Environment

(*f* **get-internal-real-time**)
(*f* **get-internal-run-time**)
▷ Current time, or computing time, respectively, in clock ticks.

c **internal-time-units-per-second**
▷ Number of clock ticks per second.

(*f* **encode-universal-time** *sec min hour date month year* [*zone*$_{curr}$])
(*f* **get-universal-time**)
▷ Seconds from 1900-01-01, 00:00, ignoring leap seconds.

(*f* **decode-universal-time** *universal-time* [*time-zone*$_{current}$])
(*f* **get-decoded-time**)
▷ Return second$_0$, minute$_1$, hour$_2$, date$_3$, month$_4$, year$_5$, day$_6$, daylight-p$_7$, and zone$_8$.

(*f* **short-site-name**)
(*f* **long-site-name**)
▷ String representing physical location of computer.

({*f* **lisp-implementation** | *f* **software** | *f* **machine**} - {**type** | **version**})
▷ Name or version of implementation, operating system, or hardware, respectively.

(*f* **machine-instance**) ▷ Computer name.

Common Lisp Quick Reference Revision 145 [2018-01-15]

LaTeX source: http://clqr.boundp.org